Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Lauri Hukkanen

Adopting Continuous Integration – A Case Study

Master's Thesis
Espoo, 2015

Supervisor: Professor Casper Lassenius
Advisor: Juha Itkonen D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Lauri Hukkanen

Title:
Adopting Continuous Integration – A Case Study

Date: 2015 **Pages:** vi + 83

Major: Software Engineering **Code:** T-76

Supervisor: Professor Casper Lassenius

Advisor: Juha Itkonen D.Sc. (Tech.)

The practice of CI is fairly new. Although its benefits and costs have been documented to some degree in the academic literature, companies still struggle with the practicalities of adopting a CI process. CI promises a competitive advantage for companies using it: the ability to react faster to market demand, and to reduce times-to-market of software products. Although a promising prospect, especially larger projects sometimes report difficulties in achieving this benefit.

This study is a case study on CI adoption and use in an industrial setting. The case context is a telecommunications software product development project at Nokia, a large Finnish company operating internationally.

An observational method and thematic analysis are utilized to gather rich qualitative insight. In addition, the study focuses on broken CI builds as concrete, measurable manifestations of experienced problems. Data gathered from a CI system is combined with expert analysis of causes for build failures to provide measured evidence of the existence and effects of some challenges.

As results of this study, challenges that the case organization faces in its CI use and adoption are identified. The case organization experiences difficulties related to automated testing, infrastructure, communication, dependency management and CI practicalities. Reasons behind these challenges, and the inherent interrelations between them are explained.

Further results include measurements from the CI system. The measurements underline the importance of CI and test environment set-up: the case personnel suffer from extensive disturbances to their CI process caused by environment set-up and maintenance. Test environments cause significant complications, while testing in these environments reveals few human mistakes in programming work.

Keywords: continuous integration, CI, case study, process improvement, software, agile software development, ASD

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Lauri Hukkanen		
Työn nimi:	Jatkuvan integraation omaksuminen – tapaustutkimus		
Päiväys:	2015	**Sivumäärä:**	vi + 83
Pääaine:	Ohjelmistotuotanto	**Koodi:**	T-76
Valvoja:	Professori Casper Lassenius		
Ohjaaja:	TkT Juha Itkonen		

CI on verrattain uusi käytäntö. Sen hyötyjä ja haittoja on tutkittu jonkun verran, mutta yrityksillä on edelleen vaikeuksia omaksua käytäntöä omiin prosesseihinsa. CI:n väitetään olevan kilpailuetu yrityksille: se auttaa mukautumaan markkinoihin nopeammin ja vähentämään ohjelmistokehityksen läpivientiaikoja. Tämän etulyöntiaseman saavuttaminen on yrityksille erittäin tavoiteltavaa. Kuitenkin, etenkin suuremmissa projekteissa CI:n käyttöönotto luvattujen hyötyjen lunastamiseksi on koettu haastavaksi.

Tämä on tapaustutkimus CI:n omaksumisesta ja käytöstä ohjelmistoteollisuudessa. Tutkimus on tehty telekommunikaatioympäristössä, kohdistuen erääseen tuotekehitysprojektiin Nokialla.

Tutkimuksessa tunnistetaan haasteita, joita tutkitulla organisaatiolla on CI:n käytössä ja käyttöönotossa. Pääasialliset tiedonkeruu- ja analyysimenetelmät ovat havainnointi ja temaattinen analyysi. Näiden lisäksi tutkitaan epäonnistuneita ajoja CI-järjestelmässä. Näistä saadaan erilaisia mittauksia ilmentämään havaittuja ongelmia käytännössä. Asiantuntija-arviot epäonnistuneiden ajojen syistä yhdistettynä mittauksiin CI-järjestelmästä antavat mitattavia todisteita erinäisistä ongelmista ja niiden vaikutuksista.

Tutkimuksen tuloksena syntyy kokonaiskuva kohdeorganisaation CI-käytön ongelmista. Ongelmat liittyvät korkealla tasolla automaattiseen testaukseen, laitteisto- ja ohjelmistoinfrastruktuuriin, ihmisten väliseen viestintään, riippuvuuksien hallintaan ja CI-käytäntöihin. Tulosten yhteydessä esitellään tarkempia kuvauksia haasteiden perisyistä ja niiden keskinäisistä suhteista.

Lisäksi tuloksiin sisältyy mittauksia CI-järjestelmästä. Näiden mittausten perusteella teknisten CI- ja testausympäristöjen hallinta on tärkeää CI-prosessille. Ympäristöistä ja niiden ylläpidosta aiheutuu tutkimusorganisaatiossa laajoja häiriöitä CI-prosessiin. Ympäristöissä suoritetussa testauksessa paljastetaan melko vähän ohjelmointivirheitä suhteessa siihen, kuinka usein ympäristöt itse aiheuttavat ongelmia.

Asiasanat:	jatkuva integraatio, CI, tapaustutkimus, prosessikehitys, ohjelmistokehitys, ketterä ohjelmistokehitys, ASD
Kieli:	Englanti

Acknowledgements

This thesis study has been granted a very wide access to interesting data at Nokia Networks, and much cooperation from their side. Especially Sampsa Tuokko and Rauno Kosamo have been instrumental in providing valuable information, pointers for research, and support in gathering the thesis data corpus. Warmest thanks to both of you and the company.

I would also like to thank the Need for Speed (N4S) research program and its administration for providing an inspiring forum for this research to take place.

Finally, I wish to thank my supervisor, instructor and colleagues for all the support they have given me during the making of this thesis. I am especially grateful to Casper Lassenius, Juha Itkonen and and Eero Laukkanen for their input and encouragement.

Espoo, 2015

Lauri Hukkanen

Contents

Chapter 1

Introduction

Agile and lean ideologies have become commonplace in software development. Rodríguez et al. [2012] found in a large study of the Finnish software industry, that 58% of companies report using either agile or lean methods. Other studies have found agile penetration rates from 32% [Begel and Nagappan, 2007] to as high as 88% [VersionOne, 2014]. It can be said that agile software development has become the norm rather than the exception.

Agile software development focuses on the importance of people, feedback and the ability to react to change [Dybå and Dingsøyr, 2008]. Lean software development applies the principles of lean management and lean product development to agile software development. The lean ideology builds on principles such as eliminating waste, constant learning, late decision making, fast delivery, and empowerment of developers [Poppendieck and Poppendieck, 2003].

Agile and lean methodologies aim to provide tangible value to the customer quickly, in the form of working software delivered in a timely manner. These ways of working have become a competitive advantage for software companies. In a survey study regarding agile development, VersionOne [2014] reports accelerated time to market to be the most often cited reason for companies choosing to adopt agile development. Also, they report that 73% of their respondents felt that agile projects were faster to complete than projects that utilized non-agile methods.

Continuous integration (CI) is a software development practice where developers integrate their code early and often. It aims to make integration of software by different developers a constant occurrence, eliminating the need for a lengthy and unpredictable integration effort near to a planned release. Developers enjoy faster feedback of their changes through the high degree of automation in CI. [Fowler, 2006].

As a result of faster feedback and a more fluid integration process, the

ability to deliver releases of software faster is often cited as a major benefit of CI [Goodman and Elbaz, 2008; Leppänen et al., 2015]. In a highly competitive market, CI gives a competitive edge in the ability to release software faster. Market share may be defined by which company gets their product out to the market quicker, making ability to release software fast a key business driver.

The benefits and drawbacks of continuous integration have been studied in some depth, although the field of CI research is still relatively young. Discrepancies in terminology and implementation of the process between different cases are prevalent. To understand the circumstances where the reported benefits and challenges of CI are realized, more in-depth case studies are still called for. [Ståhl and Bosch, 2014b]

This study sets out to research CI adoption in an empirical setting. To this end, this study focuses on CI use in a practical real-life setting, trying to discover and explain the challenges that the case project faces in their daily CI operation. The chosen case organization is an interesting subject for studying CI adoption because it is in a transitive state. During this study, the case development organization is in the process of improving their CI practices. A major driving goal for this improvement is the wish to ultimately shorten the release cycle of developed software. The following research questions are set for this study:

- *RQ1 How does the studied project use CI?*
- *RQ2 What are the challenges to efficient CI use in the project?*
- *RQ3 How do the challenges affect development?*
- *RQ4 Can the effects caused by these challenges be quantified by measurements from the CI system?*

The study has been performed as a descriptive and explanatory single case study [Yin, 1994]. The case study utilizes observations of integration meetings and collecting CI data as its main data collection methods. Qualitative insight of experienced challenges is developed by means of thematic analysis [Patton, 2002, p. 452] of the observed meetings. This insight is complemented with measurements taken from the CI system. Additionally, the case developers made an effort to categorize the reasons behind some failed builds. The categorization helps better understand the effects of certain kinds of failures.

The results of this study give an account of CI use in a practical setting. The case project faces several challenges in CI use, mainly related to software testing, dependency management, infrastructure, communication and

CI practicalities. The experienced challenges are highly interconnected and affect software integration adversely.

The rest of this document is divided in six chapters, 2 through 7. Chapter 2 concentrates on the subject background by introducing the concept of continuous integration. Because the research objective of this study is centered on the challenges of CI adoption, a brief synthesis of such challenges in previous literature is gathered in Section 2.5. Chapter 3 explains the chosen research methods of this study. Chapters 4 and 5 document the results of the study. In Chapter 4, concentrating on the results of thematic analysis, challenges that the case project faces in CI use are explained in detail. Chapter 5 provides further evidence for adverse effects to the project, by reflecting qualitative learnings about the challenges in CI use against measurements from the CI system. Finally, Chapters 6 and 7 offer reflection and concluding remarks.

Chapter 2

Background

Fast feedback cycles are a cornerstone idea of agile and lean software development. In this wider context, the benefits of fast feedback can be observed on many levels. Looking at the software product as a whole and its requirements, releasing working software to the customer early and often reduces the gap between understanding where the value for the customer lies and how the development organization interprets it. Another kind of fast feedback loop benefits the software developer in their daily work: by enabling faster feedback in testing and integrating code, developers can find problems sooner and can react upon them quicker. This enables them to correct any errors in their code while they still have it in close memory, increasing their productivity. Less context switching between development tasks is needed as feedback gets faster. Continuous integration (CI) is an enabler for these benefits on both levels – faster delivery to the customer and quicker feedback about code defects.

This chapter takes a look at the practice called continuous integration as has been written in previous literature. Section 2.1 explains briefly the history, reasonings and practicalities of CI. Section 2.2 presents a notation for describing automated CI implementations in detail. Continuous delivery, an extension of the CI practice towards more frequent and automated delivery of software, is explained in Section 2.3. The chapter concludes with Sections 2.4 and 2.5, which describe claimed benefits and challenges of continuous integration. The treatise on challenges, in Section 2.5, is of special importance to this study, as it reflects the research problem directly.

2.1 Continuous Integration

Continuous integration is a software development practice where developers integrate their code early and often, to reduce the workload and errors involved with larger, deferred integrations. Frequent and fast integration allows for faster feedback about the effects of newly created code on the whole system. This in turn makes solving any newly created problems easier, right as they are introduced. [Fowler, 2006]

Beck [2000] introduced the practice in its modern form as part of the Extreme Programming software development methodology. More than fifteen years have passed since, and today continuous integration has become a widely adopted practice in software development [e.g. Kurapati et al., 2012; Rodríguez et al., 2012].

Nightly builds can be thought of as a predecessor to continuous integration. A nightly build means that each night an automated system checks that code added throughout the day compiles successfully. [Humble and Farley, 2010, p. 65] The idea behind the nightly build and CI is the same: maintaining software under development in a working state and detecting integration errors early. However, continuous integration takes the idea further, reducing the time developers need to wait for feedback on their new code's integration success to as fast as possible. [Fowler, 2006]

In practice, continuous integration is achieved through a high level of automation of code compiling and testing activities. All changes made by the developers to the software should trigger an automated process to verify that the new changes successfully integrate with all existing code. Developers regularly check in their work to a central version control system, which is monitored for changes by the CI system. For each incremental piece of code, the CI system automatically checks the integrity of the software against preset criteria, and gives feedback on whether the increment meets the criteria. Thus, the CI system enforces quality of all changes made to the software, and alerts developers to issues right as they occur. [Humble and Farley, 2010, p. 55-65]

Continuous integration can also be extended to the point where software that passes all tests is automatically deployed to production. This extended case is an example of *continuous delivery* [Humble and Farley, 2010, p. 3-4].

Regardless of the fact that CI relies heavily on specialized tools and automated testing, it can not be thought of simply as a set of technical tools. Successful adoption of CI requires conformance to certain practices and principles. Adopting the practice of CI means more than using CI software.

To better discuss these practices, technical implementation and the con-

cepts of a *CI build* and a *build pipeline* are first introduced in Sections 2.1.1 and 2.1.2. After these introductions, Section 2.1.3 discusses some important practices according to the CI literature and manuals.

2.1.1 Technical Implementation

A minimal implementation of a CI system has two logical components: a method to execute automated workflows, or *CI builds*, and a means to give feedback on the results of the workflows [Humble and Farley, 2010, p. 63]. As a simplified example, let us consider the following workflow. As code increments are delivered to version control, the CI system monitors these changes and triggers an automated workflow, the CI build. First, a CI build might compile source code to an executable binary. To verify the quality of the compiled code, the CI system then could run unit tests on it. After these actions, the CI system reports on their success by e-mail to people who originally committed the triggering code changes to version control. Such simple verification as seen in the example is often inadequate for practical use. In practice, more automated testing is performed, aiming at increased end-to-end confidence in quality. Automated integration and acceptance tests and static code analysis are often included in actions taken by the CI system.

Although the technical implementation of continuous integration could be done with custom scripting, there are numerous software products readily available for implementing CI. Bamboo[1], BuildBot[2], CruiseControl[3], Go[4], Hudson[5] and Jenkins[6] are popular examples. There are benefits to choosing a proven product instead of a custom setup. In a study on the practices of successful open source projects, Halloran and Scherlis [2002] notes that the use of de-facto standard tools reduces the needed learning load of developers. New developers may have been familiarized with de-facto tools previously. Also, the learning effort that they invest in de-facto standard tools is likely to be valuable in later projects as well. In addition to this benefit, a CI system utilizing a ready-made product also most likely takes less effort to implement and maintain than a custom solution.

An aspect where CI implementations in practice often vary is the mechanism for communicating the status of builds. Examples range from the widely used e-mail build reports and status screens within workspaces to

[1] `https://www.atlassian.com/software/bamboo` – proprietary license.
[2] `http://buildbot.net/` – open source.
[3] `http://cruisecontrol.sourceforge.net/` - open source.
[4] `http://www.go.cd/` – open source.
[5] `http://hudson-ci.org/` – open source.
[6] `http://jenkins-ci.org/` – open source.

more exotic and subtle feedback mechanisms such as ambient lighting or small computer controllable figurines [Downs et al., 2012]. A dedicated communication channel for build status is beneficial, so that the information won't disturb other channels, and on the other hand won't get lost among other information [Downs et al., 2010].

2.1.2 Speeding Up Builds: The Build Pipeline

A CI build needs to be fast to provide timely feedback during development. On the other hand, testing should be comprehensive to give confidence on code quality. [Ståhl and Bosch, 2014a] These two goals are in conflict, because adding more tests increases the run time of the build. Many techniques have been proposed for improving build run times, some of which will be introduced below.

A good way to speed up builds is to split the CI build into multiple stages. A *build pipeline* is a collection of CI builds that run in sequence. Together, these builds perform comprehensive verification of code, but alone they are fast enough so that feedback time to developers is kept to a reasonable level. [Fowler, 2006] If any build stage fails, the complete pipeline is considered failed. All stages of testing must succeed before a software artifact is considered successfully verified. Feedback about the whole pipeline's success can be given at the time of the first detected failure. If feedback for all stages is not desired, processing resources can be saved by not running any further build stages that would not change the success state of the pipeline. Running all stages may still be desirable, even if some stage fails to verify. Unless the failure of one stage prevents the running of later stages, the test results of later stages can provide additional information about the failure to the developer. This, in turn, can help in fixing the problem.

When working collaboratively in a CI environment, testing code changes before they are submitted to the CI system for verification is important. This is usually called pre-commit testing, and happens before a developer pushes code to the version control mainline that the CI system monitors for changes. Testing code before exposing it to the CI system is important for several reasons. Changes by other developers must be taken into account. There may have been changes by other people between the time when code was first checked out for developing, and the time when it's ready for merging back into the mainline. A developer should check that their code works with the most recent changes before submitting their own additions to the CI pipeline. Another reason to perform pre-commit testing is to minimize the disturbance caused by broken builds to other people. If some testing is done before submitting the changes to CI, less broken code will be submitted

to the CI pipeline, leading to fewer broken CI builds that affect the whole team. [Humble and Farley, 2010, p. 66-67]

Complete test suites for software tend to be too large for running locally before each commit. They may also include some tests that require special testing environments, and may not be able to run on the developer's environment. However, at least a reasonable, fast subset of tests should be run. [Humble and Farley, 2010, p. 60-62] Because it is important to keep pre-commit tests fast, methods have even been developed to automatically select the most relevant tests to be run based on the changes performed or testing history. [Yoo et al., 2011; Elbaum et al., 2014]

2.1.3 Practices and Principles

Continuous integration is not only about installing a tool and taking it to use. Rather, it is a fully fledged engineering practice that requires discipline and a specific mindset. Certain principles are proposed as best practices, which help better realize the claimed benefits of CI. Some key principles are introduced below, in order to demonstrate how implementing a CI process is at least as much about practices as it is about technicalities.

Integrating often. Continuous integration equals integrating often. This requires developers to get accustomed to synchronizing their code with others at least daily, and preferably even more often. The longer the code of a developer stays isolated from the rest of the code, the more effort is required to synchronize it. [Fowler, 2006]

Automation and feedback. A broken build is not necessarily a bad sign for a project. The CI system provides a safety net to guard against errors and deviations of quality from making their way into software releases. It is supposed to provide feedback of possible errors to the developers, and this feedback is given in the form of broken builds. CI is an automated system, to which some quality control tasks can be given, even saving some time of the development team by not having to perform these same checks manually. As such is the case, there are bound to be broken builds. The best effect of a broken build is bringing to light some problems, instantly as they appear. [Fowler, 2006]

Comprehensive verification. As more and more of the quality assurance measures are trusted to an automated CI pipeline, some thought should be given to whether the CI pipeline does a decent job of affirming quality. Ståhl and Bosch [2014a] note two possible caveats to be aware of. Firstly, the verification measures taken in the CI process should be as comprehensive as possible. If verification is lacking, CI can be counterproductive, in that it builds a sense of false security. A successful CI build signals completed

verification, but if not enough quality control measures are in place, the success does not in fact tell much of quality. Secondly, all actions taken by the CI system to verify a single software artifact should work on the same artifact. Partial verification steps should take care to ensure that they are increasing the confidence in the quality of the same artifact. For example, if binaries are re-compiled between two stages of testing, are we certain that the different tests are verifying the same code?

Fixing builds immediately. A broken build that goes unchecked for a long period of time, or a high number of broken builds can be a sign of trouble. To isolate problem causes to single changes in the source code, it is beneficial to generally maintain the status of CI builds successful, or to "keep the builds green". Considerable effort should be spent on fixing broken builds as fast as possible after detection. This aids in debugging, as the code is still fresh in the memory of developers. Rapidly delivered fixes also guarantee that developers, who require a clean and successful build to get reliable feedback for their work, lose minimal time in waiting for one. [Fowler, 2006]

No commits on a broken build. New code should not be delivered to the CI system when the build is already broken. If code is committed on a broken build, errors become more ambiguous and mask each other. When the build is already broken, a failed subsequent build can either be the result of existing errors or new ones. In this way, newly created problems can be hidden. The worst case scenario is that all failed builds are attributed to be caused by the original error and development continues without regard to CI feedback. Problems may start to add up over time, making the situation even harder to resolve. [Humble and Farley, 2010, p. 66] A clear focus on fixing problems right as they appear is also called for in lean management literature, with similar arguments of avoiding compounded issues. Poppendieck and Poppendieck [2007, p. 5-6, 9] describe a "stop-the-line culture", where upon detecting an error, work would stop immediately to remedy its cause, potentially avoiding further mistakes.

2.2 Visualizing Build Pipelines

Ståhl and Bosch [2014b] point out that the popularity of continuous integration has led to great diversity in how the practice is interpreted and implemented, and that studies are needed to document how CI is implemented in industry. They propose a model for visualizing build pipelines, and in later work extend upon it [Ståhl and Bosch, 2014a]. The elements of the model are presented in Figure 2.1. For a concrete example of the notation, a build pipeline is depicted in Figure 2.2.

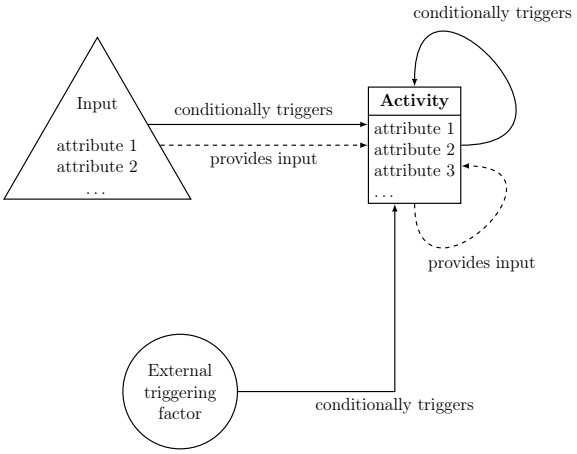


Figure 2.1: Ståhl and Bosch [2014a] notation for presenting build pipelines.

To give a sense of context, and for comparative analysis, build pipelines are depicted throughout this report using the proposed notation, with some difference in implementation. Ståhl and Bosch [2014a] used interviews as their main source of information from which to compile their representations, whereas in this report, the depictions have been composed mainly by analyzing the Jenkins CI system's configuration. Development team members have provided additional information to clarify the actions taken by the build steps. Rooting the descriptions to Jenkins configuration in this way has the benefit that each logical build activity corresponds well to a measurable build step in the CI system.

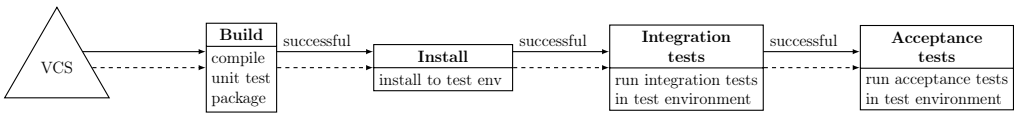


Figure 2.2: An example build pipeline described with the Ståhl and Bosch [2014a] notation.

The example pipeline of Figure 2.2 starts by monitoring changes in the version control system (VCS). All changes trigger a build pipeline, which consists of four build stages: *Build*, *Install*, *Integration tests* and *Acceptance tests*. The different stages, as actions taken by the CI system, are depicted as boxes. They run in sequence, and each of them depends on the outcome

of the earlier stages. This is depicted as the arrows connecting the boxes. The meaning of the dashed arrows is that each stage requires input data from the previous stage, which in the example case means that they rely on files created by the previous phases. The *Build* phase compiles a software executable, which is required by later stages, etc. The solid arrows represent conditional triggers of downstream actions: after each stage completes, the results of the build stage's execution determine whether the next action is triggered. In the example pipeline of Figure 2.2, downstream stages are triggered only in the case of a successful run of an upstream one.

2.3 Continuous Delivery

Continuous delivery (CD) is a practice that builds on the groundwork of CI and extends upon it. CD takes the idea of reacting to all code changes and providing feedback even further, adding an automated step of software delivery. As with CI, an automatic build and verification process takes place for each code change. Then, the verified software is prepared for deployment to an actual production environment, to instantly provide value to the end users. [Humble and Farley, 2010]

Continuous delivery is considered to be the ability to deploy developed software to the customer or end user at will, without additional overhead. Deploying each and every change in the code might not be desirable for all projects, but each and every change should result in a potential, high quality candidate for deployment. [Neely and Stolt, 2013] Thus, CD extends the fast feedback loop of continuous integration from testing developers' changes against the whole codebase to validating against needs of end users. [Humble and Farley, 2010] It can be argued, that the full potential of continuous integration is only realized when the additional step of delivery is added to it [Humble and Farley, 2010; Olsson et al., 2012].

Olsson et al. [2012] go a step further, noting that continuous delivery, in turn, is an enabler for achieving an even higher goal of an experimental and measurement-driven approach to developing software. Adopting this method, changes can be developed and released to parts of a live user base, with the measured success of such experimental small-scale releases deciding whether to deploy them to a wider audience.

2.4 Benefits of Continuous Integration

The benefits of adopting continuous integration are numerous. Debugging becomes easier, as errors can be associated to small increments in code. Fixes can be applied before a large number of errors start to accumulate, interfering with each other. [Fowler, 2006]. Risks of a large integration phase towards the end of the project are averted, as software is kept constantly in a verified working state. Development feedback cycle times are reduced and frequent deliveries become possible. [Goodman and Elbaz, 2008; Leppänen et al., 2015; Ebert et al., 2001] By merging code frequently, less merge conflicts arise, as individual code merges are less complex [Neely and Stolt, 2013]. High emphasis on build and test automation can contribute to increased quality and productivity [Leppänen et al., 2015]. Also, a good automated test coverage means that new developers have a safer environment to learn in, which makes it easier for them to catch up to speed with the project [Neely and Stolt, 2013]. Ståhl and Bosch [2013] also find similar benefits to CI, concluding that CI has a positive effect on communication, facilitates parallel development and leads to more effective troubleshooting and increased project predictability.

Literature on continuous testing and compilation also supports the claim that fast feedback aids in trouble shooting and thus increases efficiency [Saff and Ernst, 2004; Muşlu et al., 2013].

The practice of CI is quite compatible with the lean notion of flow – smooth value delivery to the customer. Because a continuously integrated product is constantly maintained in a demonstrable state, customers can easily evaluate its value at any time. [Rodríguez et al., 2013]

Miller [2008] acknowledges that there are additional costs to the practice of CI, as more effort needs to be spent on infrastructure maintenance and immediate fixes to build breaks. On the other hand, he estimates that by using CI, effort is still saved overall compared to a non-CI approach, as less manual verifications are needed before committing each piece of code.

2.5 Challenges in Adopting Continuous Integration

Adopting continuous integration is not easy, nor is it free of cost. This section presents some challenges found in previous literature surrounding CI. The articles chosen for inclusion here include single and multiple case studies in industrial cases, and some literature reviews. Some of the referenced articles set out to study problems faced in continuous delivery. Continuous

integration is an integral part of continuous delivery, and as such many of the issues faced in a CD context are in fact inseparably related to CI. Those challenges found in literature that only concern the added complexities of CD I have chosen not to introduce here. The case project in this study does not practice CD, so issues related only to the additional step of delivery do not relate closely to the studied case. A summary of challenges in literature is presented in Table 2.1.

Test automation and testing technique. CI relies on automated testing, which can take considerable effort to implement at an acceptable level [Olsson et al., 2012]. Unstable tests, sometimes referred to as flaky tests, lower developers' confidence in build results. They might even cause developers to ignore build results altogether, assuming problems to lie with the flaky tests when there in fact may be real problems with the code. [Neely and Stolt, 2013] The existence of manual testing steps may prove problematic, as it slows down the process of producing verified increments of software. Manual testing can also be less methodological, relying on individuals conforming to practices. [Neely and Stolt, 2013] Especially when working close to hardware, frameworks for adequate automated testing may be lacking, forcing the need for some manual testing steps [Debbiche et al., 2014].

Communication. Efficient communication is crucial for successful teamwork. In a CI environment, communication is an essential requirement. Effectively using the CI system and producing a working, integrated software product are both collaborative efforts that require efficient communication. The effects of lacking communication on the collaborative aspect of software integration are highlighted by Damian et al. [2007]. In studying reasons behind failed CI builds, they have traced failure back to communication gaps between developers whose code was involved in the broken integration. Another kind of communication takes place between the developer and the CI system. The CI system communicates with the developers in the form of build result feedback. Developers need to be aware of this feedback, and actively follow it for it to have any value to them. Also, the produced feedback should be understandable to the developers. [Ståhl and Bosch, 2014a]

Distributed development. Distributed development further widens the communication gap between developers [Miller, 2008; Herbsleb and Mockus, 2003]. In general, information flows less efficiently between distributed sites than it does within a single site, making it harder to find the relevant people for handling specific issues. Differing development processes at different sites can complicate the overall process of delivering software [Olsson et al., 2012]. Also, there is lower visibility to the current status and plans of distant sites. [Herbsleb and Mockus, 2003]

Management support. Obtaining sufficient management support for CI

or CD adoption is underlined in several sources as crucial to the success of adoption [Neely and Stolt, 2013; Leppänen et al., 2015; Debbiche et al., 2014; Claps et al., 2015; Olsson et al., 2012]. Clear goals for the adoption are seen as a necessity [Neely and Stolt, 2013; Debbiche et al., 2014]. Without clear goals shared between top management and development, expectations will be misaligned, leading to conflicts in perceived progress. As a consequence, developers may experience increased pressure to comply with unreasonable expectations about the speed of adoption. [Debbiche et al., 2014] Assuring management support and setting and communicating clear goals have been long recognized to be important for the success of any software process improvement initiatives, not only CI ones [Stelzer and Mellis, 1999].

Change resistance. Old habits are hard to change. Developers may feel sceptical about the benefits of CI. The process of CI requires developers to submit code for others to see earlier than they might be used to, as changes are constantly integrated. The increased transparency in programming work can feel threatening to developers. Developers may feel hesitant to expose their code early on, in fear of being judged for its unfinished state. [Debbiche et al., 2014] Leppänen et al. [2015] also note resistance to change in management to prove problematic, possibly preventing the success of process improvement initiatives such as adopting CI.

Architecture. Software architectural decisions may prove to be a hindrance to efficient implementation of CI. To give an example, tight coupling of components may necessitate building all parts of a large system from scratch during CI builds, instead of building a small subsystem more closely related to the changes made. This in turn increases build time and thus limits the possible frequency of builds. [Bellomo et al., 2014]

Sufficient hardware. Running CI efficiently requires a lot of hardware resources. Quick build times are of importance to CI, and hardware directly affects build times. [Miller, 2008; Beck, 2000; Claps et al., 2015] Lack of available hardware may prevent optimization or improvement of the CI process. As more and more quality measures are implemented in the CI pipeline, more hardware resources are needed to run them. [Miller, 2008]

Tool issues. As well as hardware, software tools are needed to perform the automated tasks involved in continuous integration. These tools may present some problems. The maturity of available tools for continuous integration and automated testing is brought to question. Some tools are ridden with bugs or they are not able support the CI process well. [Debbiche et al., 2014; Olsson et al., 2012] Debbiche et al. [2014] find tools lacking especially in the areas of version control, the build framework and code reviews. In their studied case, immature tools caused software integration to take a longer time, preventing integrations at desired frequency. Olsson et al. [2012]

report added work effort in constantly learning new tools, and the need for additional testing to verify uncertainties about how third party tools work.

Domain applicability and size or complexity of the product. Existing development processes in companies may not be suitable for the practices of CI, especially in companies that are working in close relation to hardware [Debbiche et al., 2014; Olsson et al., 2012]. Large size and high complexity of the product being developed may make CI process adoption difficult [Debbiche et al., 2014; Ståhl and Bosch, 2014a]. Also, the speed at which deliverable releases of software can be made is necessarily slower in larger projects, already because of longer compile and test times. This slows down the feedback loop. [Leppänen et al., 2015]

Understanding of the CI/CD process. A shared understanding of the process across the whole organization is needed, as well as a high degree of expertise. Initially setting up the required practices and infrastructure demands deep knowledge and investment of resources. Developers require a good grasp of the pipeline to perform their daily work. [Leppänen et al., 2015] Developers' lack of awareness of the CI system can diminish its value altogether – if its feedback is not followed or its process not understood, what use is it? [Ståhl and Bosch, 2014a] Furthermore, management and development goals need to be aligned, and this requires that the managers also understand the processes of CI and CD. [Debbiche et al., 2014]

Infrastructure maintenance. Maintenance of the technical environments required for CI can be demanding. Development, testing and production environments need to be consistent and as identical as possible to ensure that testing in different environments behaves similarly. Maintaining good conventions for the configuration of a multitude of different environments is a challenge. [Leppänen et al., 2015]

Requirement definition. Debbiche et al. [2014] observed in a case study that requirement definition for software became more difficult when a CI process was adopted. Requirements had to be broken down to smaller increments to allow for more frequent integrations, which added to the amount of prioritizing and decisions that needed to be done for each requirement. With smaller requirements, a desire to deliver visible value to the customer raised the question, whether very small increments were worth integrating at all.

Dependency management. Because the process of CI requires that code is integrated in small increments, the number of integrations necessarily is high. Multiple developers contribute to the integrated software product in parallel instead of a single effort to integrate components of code. This makes management of dependencies and coordinating their integration more difficult. Component interfaces need to be more clearly defined. Sources of integration errors may be harder to track because of more developers contributing

to the integration of a software product. [Debbiche et al., 2014] Kerzazi et al. [2014] find especially transitive dependencies to be often the reason for broken CI builds. Because transitive dependencies are not explicitly specified for a software module, the developer is less aware of their effects.

Challenge	**Nature**	**Papers**								
		Miller [2008]	Damian et al. [2007]	Claps et al. [2015]	Olsson et al. [2012]	Neely and Stolt [2013]	Leppänen et al. [2015]	Bellomo et al. [2014]	Debbiche et al. [2014]	Ståhl and Bosch [2014a]
Communication	Social	X	X							X
Distributed development	Social	X			X				X	
Management support	Social			X	X	X	X		X	
Change resistance	Social						X		X	
Understanding of the CI/CD process	Technical / Social						X		X	X
Sufficient hardware	Technical	X		X						
Test automation and testing technique	Technical			X	X	X	X		X	
Infrastructure maintenance	Technical						X			
Software architecture	Technical							X		
Tool issues	Technical				X				X	
Dependency management	Technical								X	
Requirement definition	Technical								X	
Size or complexity of product	Domain						X		X	
Domain applicability	Domain				X				X	

Table 2.1: CI adoption challenges in literature.

Chapter 3

Research Design

This chapter explains the chosen research methods of the study. The main objective of this study is to understand barriers to CI use. The study has been performed as a descriptive and explanatory case study in a software industrial context according to the ideas of Yin [1994] and Runeson and Höst [2009]. It utilizes an observational method and thematic analysis to gain qualitative insight, and explores the implications of qualitative findings by combining them with quantitative data.

The rest of this chapter is arranged as follows. Section 3.1 formulates the research problem and specific questions for this study. Section 3.2 introduces the case study method in general, and Section 3.3 describes the context of the studied case. The final two sections of this chapter describe the utilized methods and practices of data collection, in Section 3.4, and analysis, in Section 3.5.

3.1 Motivation And Problem Formulation

Although continuous integration is a widely adopted practice in the software industry, and its effects have been documented to some degree, understanding of it is still not complete. Ståhl and Bosch [2013, 2014b] have found that there is disparity in the implementation and interpretation of what CI is, both in practice and in the literature. They call for studies to document in detail the wide diversity of CI implementations, for better understanding of the circumstances under which benefits of CI are realized. They also point out that unobtrusive ways of obtaining quantitative data from CI systems would be valuable. [Ståhl and Bosch, 2014a] This study contributes a detailed account of CI use in a practical software industry context, gathering data over several months. Use of CI data to quantify qualitative findings is also

explored.

This study sets out to find challenges in the adoption of continuous integration. To achieve this, we take a practical approach, studying the use of continuous integration as it is being adopted in a real project. The main objective of this study is to understand barriers that the case project faces in CI use, with a lesser objective of describing CI use in detail. Understanding how exactly CI is used is vital to finding out possible problem causes.

Inspired in part by Kerzazi et al. [2014] and Miller [2008], we explore broken builds. Broken builds are a concrete, measurable manifestation of possible problems in the CI process. As such, they are a fruitful focus point for research. By relating qualitative learnings about what hinders CI use against data from the CI system, we gain additional evidence of the existence and extent of integration challenges in the project.

To meet the goals of this study, the following research questions have been set:

- *RQ1 How does the studied project use CI?*
- *RQ2 What are the challenges to efficient CI use in the project?*
- *RQ3 How do the challenges affect development?*
- *RQ4 Can the effects caused by these challenges be quantified by measurements from the CI system?*

To answer these questions, several kinds of data have been collected and analyzed. The collected data includes recorded observations of daily stand-up meetings on the subject of CI, build statistics and configuration information from the CI system, and categorizations of build failures based on developers' expert analysis. In addition to these main data sets, some additional background information has been gathered to provide context for the study. Background interviews regarding the use of CI in the case organization had been done by other researchers in February and March 2014, before the beginning of this research. Informants in the case company provided process documentation of the CI practices in the studied case. Together, the background information and collected data sets complement each other to answer the research questions:

RQ1 is of a descriptive nature. Background data provides a starting point to understanding CI use in the case. The chosen data collection method of continuous in situ observation allows for deep understanding of practicalities.

RQ2 aims to identify the experienced challenges in CI use that the case company faces. As part of thematic analysis of observed stand-up meetings,

a general understanding of the challenges that the participants experience is gathered.

RQ3 is set to explain in depth the mechanisms of how these problems affect development. For reasoning about causal inferences, deep qualitative insight is required. To achieve this, we again turn to the analysis of stand-up meetings. The discussions that took place in the meetings are rich in detail and explanations. Thematic analysis provides a framework for organizing the observations into knowledge.

RQ4 is concerned with finding measurable evidence for the existence and effects of identified challenges. To this end, categorizations of build failures by the developers are reflected against CI data measurements and also against the qualitative learnings of thematic analysis.

3.2 Case Study

The case study method originates from social sciences, but is widely used in a software engineering context as well. Phenomena in software engineering often involve people, their interactions with each other and with technology. Often context can not be easily separated from the studied phenomenon itself. [Runeson and Höst, 2009] For example, can one separate the concerns of software development and process model from each other? Where does process end and development work begin for a software developer? Yin [1994] argues that a strength for the case study method compared to other research methods is the ability to study exactly these kinds of phenomena: contemporary, ongoing issues which are hard to distinguish from their context.

In a good case study, research questions should be laid out to support the overall goal of the research. Data gathering and analysis should be planned so that they actually give answers to the research questions. This is what is referred to as a chain of evidence: the reader should be able to follow this chain of choices and argumentation from the overall research objective to the eventual evidence in data, and judge the appropriateness and validity of each of the analytic steps or decisions taken in the research. [Yin, 1994]

The case study method invites the researcher to combine different kinds of evidence, and also distinct sources, to form stronger arguments about the results. Being able to combine both qualitative and quantitative data and different forms of analysis is a natural strength of the case study method. [Yin, 1994]

3.3 Case Description

The case organization chosen for this study develops a telecommunications software product at Nokia Networks. The product has already been deployed to multiple customers, who are mobile telephone network operators all over the world. The software is used to manage self-organizing network infrastructure.

The software developed by the case organization is a modular Java application. It consists of a number of components providing shared base functionality that others rely on, and other components providing valuable features on top of the basic product. Not all installations have the same set of feature components installed. Components are of varying maturity, some just entering development during this research, and some having a longer legacy. The software has a web interface for administration, developed as a single-page dynamic Javascript application. The web interface connects to the main back-end software through a web service interface. Delivery is traditional in the sense that production environments of the software are set up at customer premises and require custom specialist labor for installations.

The product's development organization is large. Work is distributed across four sites in three countries. There is a significant time difference to one of the sites, while the other three follow a comparable work day cycle. Development follows the Scrum process. More detailed numbers of developers or teams involved can not be revealed because of confidentiality reasons.

Teams are generally responsible for one or a few modular software components, which may include either shared base components or those providing usable features, or a mixture of both. Teams do have access to all of the source code, and can in theory also develop components that fall outside their immediate responsibility. Often development and maintenance of other teams' components is left to the teams responsible for them, as they have more experience with them. Also, teams want to avoid overlapping work and disturbing each others' development needlessly.

In the case software product, a feature component typically depends on a number of shared components to build, and requires a specific runtime environment to run. Dependencies are managed through the Maven[1] build system.

Sites are somewhat autonomous to decide their detailed methods and practicalities of software development themselves. Of course, product level guidelines exist, and the effort to produce software is governed at a product

[1] Apache Maven.
`http://maven.apache.org/`

level. However, exact details of organization and practices are left for the sites to decide. For example, no single framework for automated testing has been mandated for the whole project, and instead each component uses testing tools as they best fit. Constant discussions are had between sites to share experiences and best practices, but the sites decide for themselves how they want to implement the fine details of development.

The product organization had begun to simplify and optimize their CI processes in early 2014. Because the company is in such a transitional stage in adapting to their new CI process, it is interesting as a subject case to study the adoption of continuous integration. The main goal behind embarking on a CI improvement project was to enable faster time-to-market in a competitive business environment as customer partners requested features. Among the desired improvements were to reduce code verification feedback time to developers, and to leverage more industry standard tools in the CI pipeline in favor of custom tools created for the company's CI needs specifically.

Our research focuses on the development teams of a single site. The teams at the studied site had begun a practice of daily integration stand-up meetings to pinpoint their major challenges in software integration. The existence of this practice makes the site an ideal target for studying the challenges they face in CI use and adoption.

3.3.1 CI in the Studied Project

The studied project uses a Jenkins CI system. The system is shared with all teams who develop the product. The system is comprised of two Jenkins machines. One acts as the master build server and serves the CI user interface. Another one provides additional computing resources for the running of builds. Builds are shared between the two machines, with the master server also running some builds. CI jobs are configured in such a way that they always run their builds on the same Jenkins machine. For integration and acceptance testing, the software is packaged and installed to separate test environments automatically as part of the CI process. Each feature component typically has one or two test environments, which each consist of one or more networked virtual servers.

Each software component runs its own CI pipeline to provide feedback for the developers working with that particular component. As there are multiple components, so there are also many pipelines running in parallel. The work of the individual component teams is integrated to form the complete software product through the CI pipeline. Typically, when a component's CI pipeline has finished verifying a code increment at all stages of testing, the component code is promoted to the product level. This in practice means tagging the

verified code from version control for a merge attempt. After tagging the new version, the product level build pipeline is triggered to try to merge the changes and test the integrated product together with other modular components. This process is depicted in Figure 3.1.

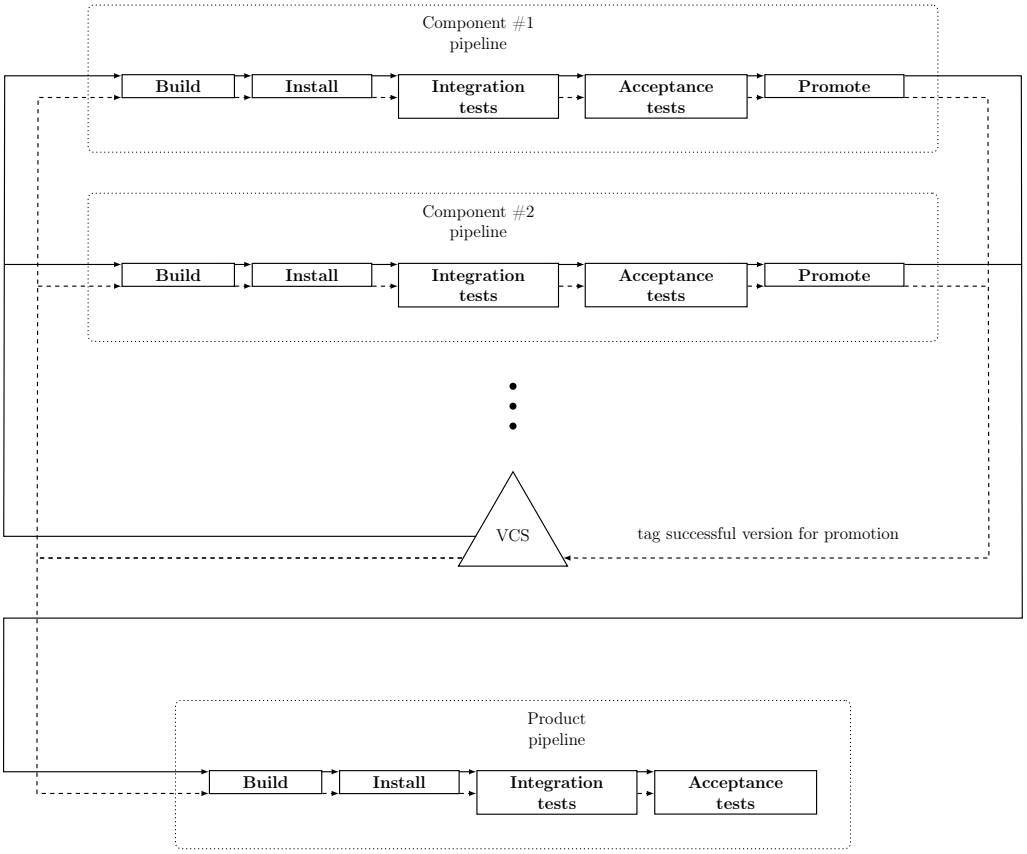


Figure 3.1: Integration of component level code into the product CI pipeline through promotion.

A standard template for CI pipelines of a component has been defined. The standard setup proposes two distinct logical CI pipelines for each software component, labeled a *fast* and a *slow* pipeline. The reason for having two pipelines for each component is to keep development feedback cycle to a minimum in the fast pipeline, while maintaining more comprehensive testing in the slow pipeline as a less frequently run task. Fast component level pipelines are run for each commit received by the version control system. Their purpose is to provide feedback during development by keeping run time at a reasonable level. The amount of testing done in the fast pipelines

is limited to keep run times under control. The fast pipeline is depicted in Figure 3.2.

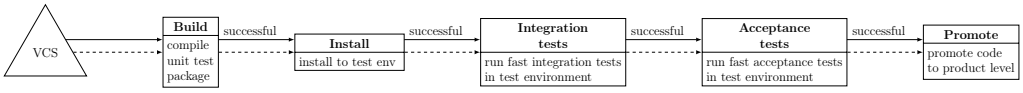


Figure 3.2: Standardized fast build pipeline for the project.

The slow pipelines for each component are run nightly. During the nightly builds, more testing and static code analysis are performed, and test environments are cleaned to a predefined state, with updated versions of base requirements. The developers refer to the reset of the test environments by the term *scratch* install. Figure 3.3 describes the slow pipeline.

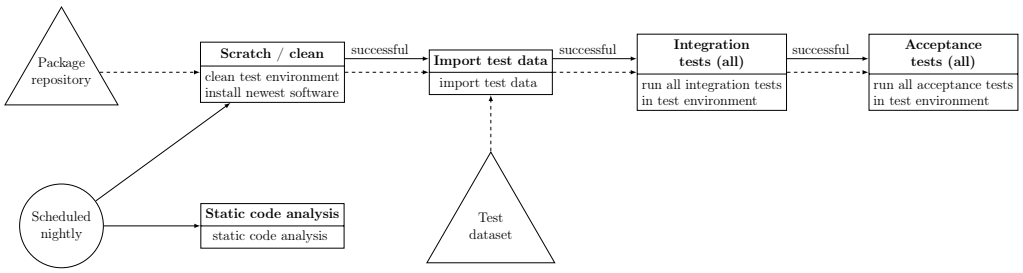


Figure 3.3: Standardized slow build pipeline for the project.

In addition to the component level pipelines, fast and slow tracks have been set for the whole product. The product level pipeline is triggered when component level code is promoted for inclusion on the product level. Promotion of code from the component to product level happens after it successfully passes all verification on the component level. As visualized previously in Figure 3.1, the product pipeline combines all promoted component code, compiles it from source and runs test suites which are a combination of all tests on the component level.

3.3.2 Daily Integration Stand-up Meetings

The developers at the case site had set up a practice of discussing integration failures across teams. This took the form of informal daily stand-up meetings, gathering participants from across all development teams on the site to discuss the status of their CI efforts. The aim of these meetings was for the

case site to understand where the major problems in their CI process lie. The stand-up meetings can be described to form a *Problem Solving Community* as defined by Tamburri et al. [2013], with its goal set to mitigate problems in CI technicalities and practices.

During these sessions, representatives of each feature component development team gathered to discuss their current status of integration and any possible problems encountered. Problems were brought forth especially if they required help across teams to solve them. The sessions would take place every day at a the same hour in the afternoon. The participants to these sessions were mainly developers, test engineers and line managers. The line managers had a dual role in also acting as scrum masters for the development teams. There was no strictly appointed team member from each team to appear in the sessions, but most often the same person would attend from each team. CI or systems specialists would attend as needed, and if they were available. The meetings were open, and other people less involved in actual development work also participated occasionally, such as project managers.

A status display of the CI system[2] provided structure to the meetings: in each session, the participants would at least discuss those builds that were in a failed state at the moment, trying to find reasons for their failure, and assigning people to look after fixing the builds. Sometimes, when failure causes were unclear to the participants or they hadn't had the time to investigate them yet, failures would be investigated on spot during the stand-up. Builds with a successful status at stand-up time might be discussed also, as participants deemed necessary. Especially recently failed ones were discussed. A team who had no failed builds at stand-up time would often still tell a bit of their status or tell of a recent failure to share information.

Usually, one person would act as facilitator for the session, seeing to it that the meeting is run in an efficient manner. The facilitator would ensure that all failed builds on the screen are discussed, corrective actions and responsibilities are noted, and most other discussion is deferred to a later time.

The meetings did not have a strict agenda. As the stand-up was a convenient time to catch the attention of a wide audience spanning multiple teams, so it was a natural place to share information between teams. Announcements of recent developments or discussions of plans were among the information shared. Comparisons were made from time to time to other sites' integration procedures, especially when facing a recurring problem in

[2]Jenkins Wall Display -plug-in was displayed at the meeting location.
`https://wiki.jenkins-ci.org/display/JENKINS/Wall+Display+Plugin`

integration. The participants would ask if other sites had the same problem, and how they would have solved it.

In conclusion, the meetings served several purposes. Systematically going through all failed builds raised awareness of build status. The stand-ups made build breaks public, perhaps increasing the motivation of participants to fix them. Problems and solutions were discussed, giving light to some wider problems and educating developers across teams. Participants shared news and understanding with each other. They asked for help and help was given.

3.4 Data Collection

Daily integration stand-up meetings were observed and audio recorded in full from June to September 2014. As described in detail in Section 3.3.2, the topic of these meetings was to discuss the problems that development teams encountered with CI and software integration in general. I was present on location to record and observe most of the meetings myself, with a few absences being covered by my colleagues.

Observations were chosen as a method partly to reduce bias that could be present in interview studies, and in part to understand the context as well as possible. Observation lets the researcher experience the context of the case first hand and aids in understanding the importance of small nuances. The method also leaves less room for subjects to choose to avoid uncomfortable topics, as compared to interviews. [Patton, 2002, p. 261-264]

The recordings from the meetings were transcribed and subjected to thematic analysis. As a result from the thematic analysis, a qualitative account of experienced challenges in the adoption and implementation of CI, in a large industrial context was gathered.

Also, quantitative data recorded from the CI server could be reflected against the insight provided by the qualitative analysis. Build data and configuration information was gathered from the studied Jenkins CI system from the beginning of June to the end of November 2014. This build data includes metadata about the builds such as start and end times, causes for triggering the builds, information about version control system changes included in the build, and also a fine-grained output logging of the build execution. This data provides time series and statistics about the builds. Various metrics, such as times to repair a broken build are calculated based on the build data.

Additionally, the developers made a high-level root cause analysis of some of their failed builds. This was done as broken builds happened, as fixing the builds required the developers to figure out a cause behind breakage in any case. They recorded the build failure root causes into the Jenkins system

through a plug-in, which allowed for linking the categorizations to all other build data.

3.4.1 Observed Stand-Up Meetings

A total of 61 daily stand-up sessions were observed at the studied site from June 5th to October 2nd, 2014. These sessions added up to over 13 hours in length.

Regardless of the many functions the stand-up meetings served, the meetings were not often prolonged. On average, a stand-up session lasted 13 minutes. The longest session lasted 39 minutes and the shortest one 4 minutes. A few shorter sessions were due to a number of participants being unavailable due to vacations or occasional scheduling mismatches. A longer session would typically include some investigation of integration problem causes during the stand-up, by scrolling through build logs together.

The stand-up meetings were audio recorded and transcribed in full. I have transcribed one third of the material myself, with the rest transcribed by a professional service. I have checked all of the material transcribed by the third party for accuracy and consistency, and made corrections as necessary.

Notes were taken during the observation of stand-ups. Taking notes was valuable for three reasons. Firstly, they provided a quick index for later reference to the documents, giving pointers to important content within each session. Second, they provided initial ideas for developing coding schemes for thematic analysis. Finally, if some part of the context of discussion was unclear to the observer, needs for clarifications would be noted. Subsequently we would discuss with the participants immediately after the stand-up to clarify.

3.4.2 CI System Data

Data gathered from the Jenkins CI system includes global and build job specific configuration, meta-data of builds, raw console logging of the execution of builds, and compressed version control system change logs for each build. The Jenkins meta-data model of a build includes data such as start time, duration, success state, triggering cause and possible originating builds upstream in the pipeline.

In addition to the aforementioned data that the Jenkins system provides as-is, information pertaining to a broken build possibly includes a developer's categorization of the reason why the build failed. The developers at the studied site had made an effort to categorize their failed builds on paper previously, to prove or disprove gut feelings about which are the most

common failure causes. The effort was extended during this study by recording the causes of failed builds within Jenkins, so that the effects of different kinds of build failures could be analyzed. The categorizations were recorded through a simple Jenkins plug-in developed expressly for gathering this data during this study. A screen shot of the plug-in can be found in Appendix A. The data gathered via the plug-in from developers includes a failure cause, an owner for solving the issue, and a free text field for additional notes, which the developers mainly used for pointing out more exact information about the build break, or for guessing at possible solutions. The benefit of collecting the failure cause analysis data within the CI system over taking notes somewhere else is that the identified failure cause can be linked with certainty to a specific build in the CI system. This, in turn, lets us compare the effects of different types of failures, by linking the categorization to other data pertaining to the broken build, such as how long it took to repair.

3.5 Data Analysis

This study embraces a qualitative approach to organizing its data into understanding. The key analysis method used has been thematic analysis. Thematic analysis involves labeling data with codes, and searching for overarching themes in the data [Braun and Clarke, 2006]. The process of coding has been open, meaning that no preset framework of codes was used to label the data. Instead, the coding scheme was iteratively developed during the study to fit and describe the data. This is called open coding. [Strauss and Corbin, 1998, p. 101-121] This section explains the principles of these methods of data analysis as they have been applied to this study.

3.5.1 Thematic Analysis

Thematic analysis is a form of qualitative analysis to find themes, patterns and categories that are present in data. It is a form of reducing data to identify meanings within [Patton, 2002, p. 452]. The method is flexible with no strict framework of application, and as such is widely applicable to different circumstances for providing rich and complex insight about data [Braun and Clarke, 2006].

Braun and Clarke [2006] have given guidelines to thematic analysis, describing a process involving six phases. The phases are listed in Table 3.1. These guidelines are generic in nature and are not meant to limit the inherent flexibility of thematic analysis as a method, but work well in describing the analytic process involved. The thematic analysis of stand-up observations in

this study has followed these steps.

1. Familiarizing with the data:	Transcribing, reading and re-reading data, making notes on initial ideas.
2. Generating initial codes:	Coding interesting features of the data in a systematic fashion, collating data relevant to each code.
3. Searching for themes:	Collating codes into potential themes, gathering all data relevant to each potential theme.
4. Reviewing themes:	Checking if the themes work in relation to the coded extracts and the entire data set, generating a thematic map of the analysis.
5. Defining and naming themes:	Ongoing analysis to refine the specifics of each theme, and the overall story the analysis tells, generating clear definitions and names for each theme.
6. Producing a report:	Selection of vivid, compelling extract examples, final analysis of selected extracts, relating the analysis back to the research question and literature, producing a scholarly report of the analysis.

Table 3.1: Phases of thematic analysis [Braun and Clarke, 2006].

First, it is important for the researcher to get deeply familiar with the data. This is done by transcribing the data and repeated, active reading of it. At this phase, notes should be taken to write down some initial ideas of codifying and organizing the data. [Braun and Clarke, 2006]

After familiarization, coding of the data begins. Coding is the process of indexing the research material for later synthesis. Codes are labels that identify some conceptual or semantic feature in the data. [Braun and Clarke, 2006] Coding is an iterative process, requiring multiple readings of the research material. The first reading may well be spent in search for a coding scheme – determining what is important in the data, and how to generalize it. Subsequent readings concentrate on the categorization of data against the developed scheme, which itself still may be altered at any point. [Patton, 2002, p. 462-465]

Next after coding on the list according to Braun and Clarke [2006] are three phases which involve the development of themes in an iterative manner. The researcher starts combining codes to find overarching themes in the material. Themes, or categories of codes as some literature calls them [Patton, 2002; Strauss and Corbin, 1998], are categorizations of codes to form a more

abstract representation of the issues inherent in the data. The higher abstraction level helps to explain and generalize the original material [Strauss and Corbin, 1998, p. 113]. The theme construction phases of thematic analysis contribute the main reduction of a wide set of data into a condensed form that can be more easily grasped by the reader. A good set of themes regarding a data set should be internally consistent and externally complete, meaning that data categorized in a certain theme does relate to other data categorized similarly, and that no relevant information has been ignored in the process of coding and reduction to themes. All data should be given equal attention to avoid bias. The themes should be meaningful for the research objective and the categorization should be credible to the persons who provided the information in the original data. [Patton, 2002, p. 465-466]

3.5.2 Coding Practices

Open coding [Strauss and Corbin, 1998, p. 101-121] is an inductive form of codifying research data; it involves discovering features within the data without a preset framework to code data against [Patton, 2002, p. 453]. As open coding does not rely on pre-existing theoretical frameworks to analyze data, it is useful when searching for new phenomena in the data.

The coding of observation data was done in an open manner. The research objective sets a pretext of finding causes for broken builds, which was kept in mind throughout all the observation and coding. Additionally, while observing the stand-ups and discussing with the participants afterwards, notes were kept to get an idea of initial coding schemes. Other than these loose pretexts, no strict template for coding was set.

Chapter 4

Experienced Challenges

This chapter presents the qualitative findings of the thematic analysis: experienced challenges in CI use and adoption in the studied project. The challenges presented here are the significant ones that came up in the stand-up discussions. These challenge themes are mostly the ones that have occurred often or constantly in discussions during the studied time period of four months. Some of the presented themes occur less often, but are strongly interleaved with other findings, giving light to possible explanations. As one of the goals of this research is to understand the barriers to adopting continuous integration in depth, it is highly relevant for the research objective to include also explanatory topics in analysis.

A general level thematic map of experienced challenges is presented in Figure 4.1. Observations for each major challenge introduced in the figure are then discussed in the following sections of this Chapter 4. To close this chapter, Figure 4.3 gathers the observations from each individual challenge theme together, in an attempt to draw a more comprehensive overall picture of the challenges and their interrelations.

Some quotations from stand-up participants are offered to illustrate the reported issues. The given quotations are presented mostly in the form they were heard in, with some deviations from this rule. Parentheses delimit sections that I have added to provide context, such as for pronoun references linking to earlier conversation. I have translated some quotations from Finnish to English. Also, to maintain confidentiality of the people and the exact project under study, some parts, such as names of people and software components, have been obfuscated or removed. And finally, some irrelevant parts in some quotations, mainly acknowledgements from the listener or out-of-topic interruptions by other people, have been cut out for the sake of readability. A single quotation will be delimited by quotation marks and some empty vertical space. Where two or more quotations are presented

after one another, they should not be assumed to be in chronological order, or even from exactly the same conversation. The quotations are separate, distinct examples of a wider topic being presented. Connections between presented quotations are purely thematic, unless otherwise noted.

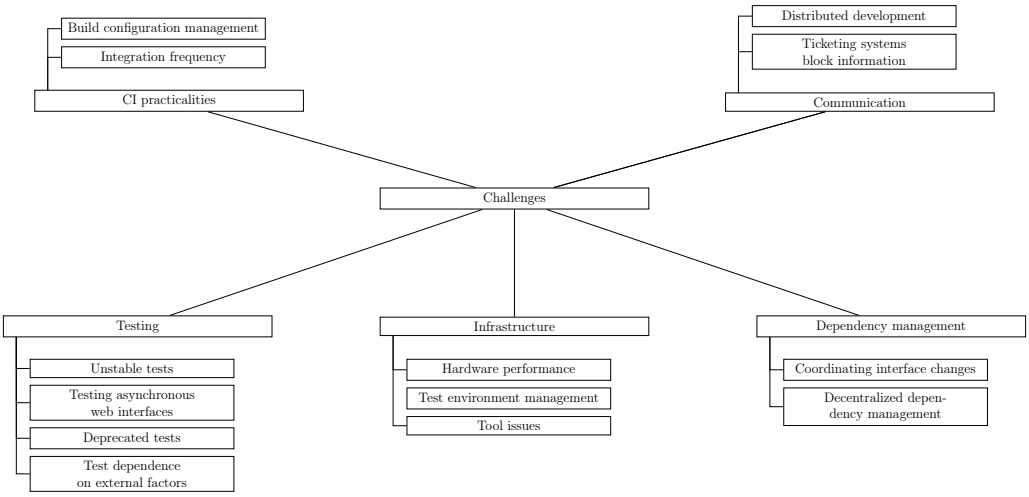


Figure 4.1: Experienced challenges as a thematic map.

4.1 Testing

Good automated testing is an essential requirement for achieving the major benefits of continuous integration. Testing complex software automatically is not a trivial task. Previous studies have already pointed out that testing is a major challenge in implementing CI [e.g. Neely and Stolt, 2013; Leppänen et al., 2015; Claps et al., 2015]. Thus it does not come as a surprise that testing issues are brought up regularly in the stand-up meetings. The major testing problems were issues with *test techniques*, especially *asynchronous testing*, the existence of *deprecated tests*, *unstable or flaky tests* and *dependences in tests to external data or environment.*

In addition to the themes described in detail, some general testing problems were often briefly noted, but not discussed in much detail in the stand-ups. Examples of such issues are small mistakes in test implementation and omissions to update tests when altering code.

4.1.1 Unstable tests

The results of unstable tests, sometimes also called flaky tests, are not deterministic. A flaky test may fail for a number of reasons related or not related to errors in the code. Its indeterministic nature confuses the developer trying to figure out its cause and fix it.

> *" These tests have been, the whole time categorized as unstable. One day all of them work, next day all of them fail. And all of them work and all of them fail. "*
> — Test automation engineer

Unstable tests were commonly cited as causes for failed builds in the project. A number of these were accountable to problems in implementation of asynchronous testing, and a part were caused by having deprecated tests in the project. These problem topics were significant enough to warrant some deeper analysis. Asynchronous testing and deprecated tests will be discussed in more detail in subsections 4.1.2 and 4.1.3, respectively. Other reasons for unstable tests also were suspected, such as changed test environment configuration or deficiencies in external test data.

> *" TA: It was working yesterday. I don't know, there hasn't been any changes so I'm expecting that there is something environment changes.*
> *LM: That sounds like an unstable test to me. "*
> — Test automation engineer and line manager

> *" It was red yesterday and nobody knew why. It had been green and it went red just, we haven't done anything because it changed but it's that problem that's going on the whole time these are, not behaving very well. And no-one knows why. "*
> — Test automation engineer

A lot of the time participants did not have a clear idea of what was the exact cause for instability in the tests, which might just mean that the team hadn't yet had enough time to study the issue in depth. However, in some cases, deeper analysis of failure causes for unstable tests could be omitted, and the instability taken for granted.

> *" How is it broken, if nobody's done anything? These are a bit like this. Sometimes they pass, and then they're red again. It's a bit vague. And nobody is going to do anything about it after all. "*
> — Developer

Efforts were made to circumvent unstable tests, such as retrying the failed tests. The studied project had used a retry mechanism for some tests which

were failing sporadically. The participants did not consider this a good practice, even though it was used. Sometimes the limit for retries was set very high.

> *" They have counters to try 40 times (...) and when it finally passes, mark the test case green. "* — Line manager

Although there is some value in maintaining a green build even when facing unstable tests, retries are problematic. The system under test may actually be unstable, in which case a real error may be masked by retrying the test until a successful result. Running the test multiple times also takes more time and hardware resources. As test run time increases, so does the feedback loop to the developer increase in length, reducing productivity. There is also a vicious loop related to hardware utilization and tests that tend to break based on assumptions in timing. When utilization increases enough to meet a bottleneck, the latency of a web server in handling requests also increases. And as the latency increases, more tests whose success depend on fast response will fail. If the tests are set to retry on failure, retries will be run, further increasing the load on hardware.

> *" Now that we know the root cause, may be that the 'try 40 times' practice was implemented as it was working to at least some degree, but it isn't at all the smartest way to do it. "* — Line manager

4.1.2 Testing Asynchronous Web Interfaces

There were some reoccurring difficulties in the technical implementations of automated tests. Major among these was the challenge of testing asynchronous web interfaces reliably. On a number of occasions, lengthy discussions were had during the stand-up meetings about the timing and implementation of tests targeting the web interface of the developed application. These are tests to verify actions done through asynchronous HTTP requests between a single-page Javascript user interface and a back-end application providing a web service interface. Sometimes, tests assumed that the results of an asynchronous request would be immediately or very quickly available, even though the request took some time to process. If a response to the request was not received fast enough before the test proceeded to verify the response, the test would fail. Under such timing assumptions, failure may depend upon hardware load on the system under test or the CI build node running the test. In the studied project, the CI build nodes run the browsers that are used for testing, and the system under test serves the data to be

tested. High load on either system may increase the latency of processing, which may result in bad timing for the test.

> *" There were test cases that sent an action to the back end, and then proceeded (to verify the results). But the expected result was not available, because it was still being processed. "* — Line manager

Another challenge in testing dynamic web applications is that the domain object model (DOM) changes as the application is interacted with. The DOM is a data structure that holds the internal representation of the web page in the browser's memory. Testing tools utilize this data structure to programmatically access the web page for testing. References to UI elements in the DOM structure may be stored for reasons of convenience or efficiency by the author of a test or by the testing tool. Stored references to DOM elements get deprecated as interactions by the user or data from the back-end system trigger changes in the DOM. Elements may be modified, deleted or replaced, and finding out when exactly this happens and for which elements can be challenging.

> *" The Javascript response has re-created the element. It looks exactly the same, and you think it's the same, but it's not the same instance anymore. "* — Developer

> *" Regardless, when the DOM changes, it changes. If you have a reference to old elements, it's your problem. WebDriver (a component in the test framework Selenium) will tell you it no longer exists. "* — Developer

A common way to prevent test failures caused by unpredictable timing in asynchronous execution is to execute the tests in a retry loop. In the case of failure, tests would be tried again until a successful result is reached or a preset number of failed retries has passed. This may sound problematic, but indeed it is a commonly used technique. For example, the Android API contains an inbuilt method for retrying tests.[1] The case project used retry loops in trying to circumvent test failures from time to time, as explained in Section 4.1.1.

Another simple way to improve the success chance for tests that fail because of unpredictable timing, is to add a longer time delay between actions and verification. This also presents some problems. Each added delay step slows down the test and the whole CI verification process. How much delay

[1] Android API: FlakyTest annotation.
`http://developer.android.com/reference/android/test/FlakyTest.html`

is enough? If the system under test is under heavy load and responds slowly, longer delays are required, still not guaranteeing success for the tests.

> *" There were sleep statements in there — wait 30 seconds and try again. These were slowing it down. "*
> — Line manager

4.1.3 Deprecated tests

The project was weighed down with deprecated tests. Some tests were not maintained, and were in the process of being refactored to use a different test tool. For the most part, deprecated tests were originally made for the Robot[2] testing framework. These tests were highly unstable and considered hard to maintain. They were causing failed builds constantly, and as such the results of these builds provided no information to the team, and were ignored.

> *" Maybe they need to get rid of them because, so long as no-one is allowed to invest energy to fix them, and I think it is not worth to do that because we're not sticking with these Robot tests and, I don't know what to say. That's the only thing in that project that's currently troublesome. And with any luck, we get this to look green but the problem is... the problem is there. These are unstable tests, they're not... no-one is investing time to fix them so they have to decide where they're going. And actually our main testing then is elsewhere. "*
> — Test automation engineer

A component team had designated one build job in their CI pipeline for storing these deprecated tests. The team themselves referred to this build as the *'trash can build'*, *'garbage dump'* or *'cemetery for broken Robot tests'*, and expected it to fail always or almost always. Discussions were had about disabling the build or removing these tests so as not to distract developers and needlessly produce red builds on the CI server.

> *" I don't know is somebody even trying to fix that, because it's Robot tests, which we have not been taking a look at for a long, long time. "*
> — Developer

> *" But actually this, I think is the cemetery for these broken Robot test cases. "*
> — Line manager

The team was not investing time into fixing these tests, so the general consensus was to outright remove them. However, these same tests remained

[2]Robot Framework.
`http://robotframework.org/`

a topic in discussions throughout several months. The build provided no information and nobody acted upon it, but the stand-up protocol of discussing all red builds on the status screen at stand-up time reminded the participants of these deprecated tests constantly.

> *" DEV: It's the one that should be cleaned out.*
> *PM: Why is it like so? You do not grow tired of seeing it red?*
> *DEV: I don't know. It was already disabled once. "*
>
> — Developer and project manager

4.1.4 Test Dependence on External Factors

Some problems were encountered with automated tests being dependent on external test data or environment state. Integration and acceptance testing in the studied project is performed against installations of the software product in designated test environments. The test environments try to emulate a production environment as closely as possible.

A test environment consists of networked virtual servers, to which a current development version of the software is installed. The software's database is populated with a large dataset when it is installed, and the tests assume that this data exists. This presented some problems for the teams. Switching from one test environment to another was a common cause of grief, as the new environment needed to be prepared with the data.

> *" DEV: When we changed the test environment for this slow build, those tests started to break. Last week I think we configured them to run in the same environment as the fast CI build. But now, it seems, the configuration hadn't been made, and the data didn't match.*
> *PM: There's no technical reason why they couldn't...?*
> *DEV: The data is different.*
> *LM: Because their test cases do not generate the data themselves, but they assume the existence of some data, which is a bad feature for test cases. "*
>
> — Developer, project manager and line manager

> *" There's some sort of test data - or some problem, that it doesn't work in this environment currently. "*
>
> — Developer

> *" We're using a new test environment, which doesn't even have data yet. "*
>
> — Developer

Dependence on specific environments could sometimes lead to long delays in fixing the builds, or even temporary disabling of parts of the build pipeline altogether for some feature components. Test environments were mainly not

managed on-site, which added additional delay to modifying or changing them.

> *" PM: What's the situation with this one?*
> *CI: The situation is that we have no test environment for it, because the tests won't work in other environments. So it's disabled. "*
>
> — Project manager and CI specialist

In addition to dependence on data, some tests made assumptions about the runtime state of the test environment, or that of a test browser. Caching within the test browsers or even in the system under test could affect results. Caches may retain some information from the previous test run that should not be there, and they may affect timings of execution considerably.

> *" The browser was not closed between test cases, and the browser cached the location where it was at the moment. The next test case assumed to get a clean browser. "*
>
> — Line manager

> *" One example is that when you first run it, the framework loads things into cache. And it may be that the request times out, that it takes half a minute before it responds to a click. And then, when you run it another time, no problem, because the framework has the Javascript code in cache. "*
>
> — Line manager

4.2 Communication

Writing software is a collaborative effort that requires working communication. Conflicts between code developed by different people are revealed when the code is integrated into one. Efficient communication is necessary for both avoiding and solving integration problems. The case project organization experiences some challenges related to communication in their effort to continuously integrate a working software product. The most notable of these challenges are *distributed development* and *inefficiencies in communication through ticketing systems.*

4.2.1 Distributed Development

The development effort of the studied software product is globally distributed, which presented some complications. The studied site is heavily dependent upon work done on other sites in ways which affect everyday development and the running and results of CI builds. Work from different sites may interfere with local work, and synchronizing development is harder across site

borders. The CI system for the project is shared by all sites developing the same product. A part of the test environments are shared between development teams, who may develop different components and are not necessarily co-located. Some software components that a large number of others depend on are developed off-site or collaboratively between sites. Also, the test environments used for both manual testing and the CI pipeline are physically located and administered on another site.

The shared CI system is a necessary tool for the project to integrate a working software product often. However, coordination challenges occur between different sites. As communication over site borders is not as efficient as it is within sites, some surprises are encountered when for example configuration is changed for a shared test environment, or when changes are made in dependencies. Dependency management was a very common and relevant problem subject in the project throughout the studied period, and will be elaborated in more depth in Section 4.3.

> *" I don't know about the latest, but, site X, they are making changes. Of course they're doing their thing, and today that's been causing all sorts of problems, because they've changed the config-files which designate, for example, the servers to connect to. And they've replaced them with their own setup, after which our tests fail. It would be really nice, if they had their own test environment where they could do their thing. We've had a bit of a crisis today. "*
>
> — Developer

> *" Product (build) failure. In site X the teams had moved some components under a new subfolder in the version control. They made their own little structure in there. And they are fixing this so.. "*
>
> — Line manager

Distributed development was a well known challenge in the case organization. To improve coordination between sites, the project personnel were starting to hold more weekly follow-up meetings between sites.

4.2.2 Ticketing Systems Block Information

Some trouble was had with the efficiency of ticketing systems to transmit required information. Tickets are seemingly lost in queues within the ticketing systems, and communication is far removed from the communicating parties. The project used two different ticketing systems, and complaints were heard of both of them about issues being stuck in the systems or not

being understood on the receiving end.

> *" LM1: Who wants to hear news about the ticket on the test environment?*
> *TA: What did you find out?*
> *LM1: That the ticket was moved from one queue to another.*
> *LM2: But the work had yet not been undertaken?*
> *PM: (ironically) But of course, moving the ticket itself is already hard work, so the work has begun.*
> *LM1: They had asked person A to do things, and transferred the ticket to team B.*
> *PM: Sounds very familiar. Moved to another queue, and lost therein.*
> *LM1: We could take a Phone-Jira there. "*
>
> — Project manager, two line managers and a test automation engineer

The participants to the stand-ups perceived a lack of urgency in responses to their tickets. They had developed the habit of calling by phone to underline the importance of their tickets and to draw attention to them, as they felt that the tickets wouldn't otherwise be given attention. The term *'Phone-Jira'* emerged to be used every now and then, to refer to a phone call whose main purpose is to raise awareness of a ticket in the receiving end. Jira[3] is of one of the ticketing systems used in the project.

> *" TA: The Jira tickets haven't changed, but–*
> *LM: Ah, yes, this is a case of Phone-Jira. One must call after the tickets. "*
>
> — Test automation engineer and line manager

4.3 Dependency Management

Dependency management was a very common source of grief in the project. One of the basic premises of CI is, of course, to expose problems in integrating dependent software components, and to make integration a *non-event* – regular and easy [Fowler, 2006]. However, there are a number of aspects in how dependencies are managed that may affect the ease of integration and keeping the CI builds green, before a non-event status can be achieved.

The major issues regarding dependency management observed in the stand-up discussions were the lack of *coordinating interface changes* and *decentralized dependency management.*

[3]Atlassian Jira.
`https://www.atlassian.com/software/jira`

4.3.1 Coordinating Interface Changes

Software interfaces between dependent components under development were in flux, and some participants felt that change management of these interfaces was not rigorous enough. Sometimes, interfaces in dependency components would be altered without the consumers of the interfaces being notified of the changes. This causes failed builds for the team consuming these interfaces, and loss of some time spent debugging the build break and fixing it.

> *" But, again still, how we change (the) public API in these projects is a bit to me questionable. Because if we have contract... Contract means, we have public methods that we publish that, people who write software on top of this base product, they use these methods. How can that method just disappear? "*
>
> — Developer

4.3.2 Decentralized Dependency Management

Dependency management in the project was quite decentralized, with components defining their compile time dependencies mostly independent of other components. Because of this, definitions of dependencies and their required versions varied across components. When integrating these components together, a so called diamond dependency problem may occur: if two or more components require the same dependency, but have defined a different version for it, which version should be used to compile the integrated component [Coutts, 2008]? This is illustrated in Figure 4.2.

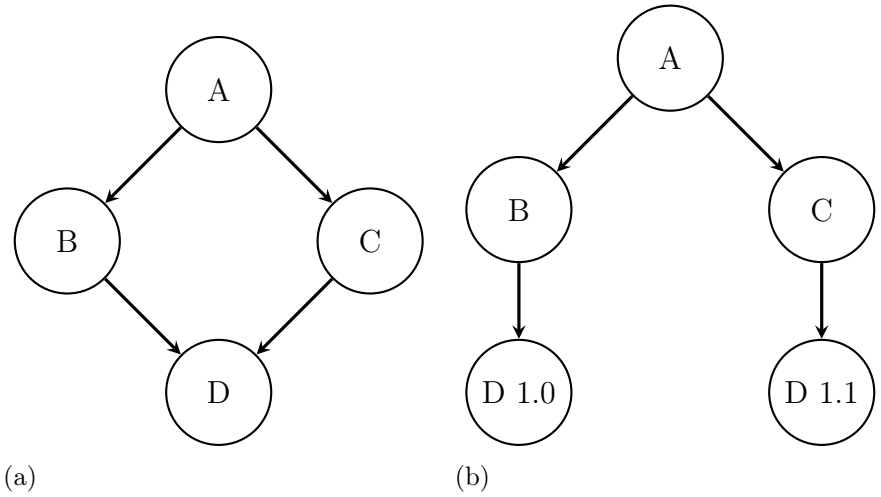


Figure 4.2: Diamond dependency problem [Coutts, 2008]. (a) A depends transitively on D through both B and C. (b) Which version of D does A require, if B requires D 1.0 and C requires D 1.1? The different versions of D may be incompatible.

Two strategies were employed to solve the problem with multiple versions. One of them was for components to define a range of compatible versions for the dependency. Ranged dependencies were handled by the build tool, Maven, which would attempt to select a single appropriate version that would match version requirements from all components included in the build. Another strategy, used for internal dependencies of the project, was re-compiling the dependency together with the dependent component, thus guaranteeing a single version of the dependency to be produced in the compiled component. Neither strategy was perfect. Re-compiling caused longer build times and excessive CPU utilization. Ranged dependencies could make it harder later on to hunt down errors, as versions of dependencies were ambiguous. Additionally, the build tool used, Maven, would take exceptionally long times in handling ranged dependencies for builds. An added nuisance was that version range definitions would have to be updated by hand.

The complicated interdependencies between software components combined with decentralized dependency management caused a need for re-compiling certain components multiple times. Re-compiling would take place on many levels. Components might require re-compilation of dependencies when building the component for testing. Again, when integrating the whole product together, a re-compilation would be required. This caused the CI builds to be slower than necessary, and for them to take more CPU time.

Computing hardware was a bottleneck for this project, in part attributable to a high amount of re-compiling of the same software components.

> *" They are not happy and we are not happy with the overall performance either, because now whenever anybody makes changes in any of these components, all of them are built. The reason for that is that they are actually, pretty much a spaghetti in the sense that they are all interdependent on each other, all these components. And because they have dependencies between each other, to get the dependencies updated you have to in practice build all of them. "*
>
> — Line manager

The project was taking action to increase centralization of dependency management, by having versions for dependencies declared once, globally, instead of at the component level. The goal was also to get rid of ranged dependencies altogether, in favor of static, automatically updated references to latest tested builds.

4.4 Infrastructure

The base infrastructure required to run CI in a large project is a rather complex combination of hardware and software. Each part of the combined system is complex in its own right and deserves careful attention. The case project experienced several infrastructure-related challenges in using CI. The major ones were *insufficient hardware*, *test environment management* and *tool issues*.

Hardware and tools need to fit their purpose well and not hinder the development work. Projects tend to grow in size over their lifespan, so hardware may not be an issue until a later point in time. Such is the case with gradual CI adaptation as well. Once the point is reached where hardware does become a problem, established practices and already set up environments can be slow to change, even if the need for change is recognized. The bureaucracy of large enterprises may offer its own problems in acquiring sufficient hardware.

4.4.1 Insufficient Hardware

The studied project had a lot of hardware issues during this research. There were insufficient resources for both the CI build servers and the test environments used to serve the application for testing. Hardware problems became critical at a point in time, where all participants to integration stand-up meetings would just cite the CI server being unresponsive as a reason for not

being able to do much anything regarding integration. Build times would be lengthened by hardware over-utilization many times over, and an unresponsive and unreliable user interface to CI would cause frustration.

> *" PM: So, this affects compiling and all?*
> *CI: Everything is, it's so stuck up, everything takes longer than previously.*
> *DEV: Our component has now been compiling for two hours.*
> *TA: For this week the whole system has been horribly slow.*
> *PM: Two hours to compile?*
> *DEV ...And counting.*
> *TA: Our component also took an hour, even though it usually takes ten minutes. "*
>
> — Project manager, CI specialist, developer and test automation engineer

> *" LM: Oh. A timeout. (referring to the CI user interface)*
> *DEV: Our explanation (for a broken build) is actually that there.*
> *LM: Well, it does slow you down if you can't access (the CI interface) to study what's wrong with it.*
> *DEV: No, but it's a proxy error.*
> *LM: Well, those will be affected in the same way. Note to the researcher, that we're having occasional over 90% CPU utilization on the server, and don't know exactly why. We already moved some builds from the server to another one, but... "*
>
> — Line manager and developer

At times the CI system's interface became hard enough to work with, so that trying to access information that should be readily available at all times caused frustration and loss of development time. This was felt as being a more significant issue than lengthened build times, as slow information is better than no access to information.

> *" I'm not against moving these (build jobs) to the other server so that one seems to have more resources. Besides actually I think big part of, I came to think of this when person A explained about how Apple had a different CPU for their user interface and a different one for the business logics. I don't think people mind actually the builds taking long, they mind the Jenkins user interface being slow to reflect, so whether the build takes 20 minutes or 25 minutes, people probably don't even notice. But they notice when the user interface is hanging. So is it more resources on the server where we have the user interface so... "*
>
> — Line manager

Increased latencies caused by high load on computing resources may expose some previously well functioning tests as unstable. There were some problems in the project in test timing and stability, discussed in some detail in section 4.1. Some number of builds that failed because of over-utilization

may be due to unreasonable assumptions in test timing. However, the hardware issues were felt very widely throughout different teams. In the worst case, hardware bottlenecks can make even reliable software behave sporadically.

> *" Our tests work every now and then. It depends on how loaded the machine is. We haven't really done anything there that would break it. "* — Developer

High computing load was a major blocker for CI in the project at times. To get around the issue, some CI build jobs had to be halted every now and then to preserve resources for other builds. This in effect means taking some components out of the CI loop, decreasing their integration frequency.

Two factors in the studied process and organization contributed to bottlenecks in hardware. Firstly, even though the project had two CI build servers to work with, there were times when builds and development teams were suffering of over-utilization on the one server, while the other machine had very low CPU usage. Under-utilization of available hardware led to over-utilization of part of the hardware. Teams were themselves responsible for configuring their CI builds, with a few specialists helping them. Developers did not feel confident enough to do all CI configurations themselves. The teams avoided re-configuring other teams' build jobs, so with many teams involved, it could take a while before action was taken to spread the build load among servers. Also, the project was geographically distributed, which caused additional delays in communicating, finding the right people to help and coordinating required changes.

> *" DEV: This server is totally stuck again.*
> *PM: I've asked person A today (at another site X), that they'd take all product level integration builds and these other component builds off. He mailed person B (also at site X), who is on vacation, but answered that some of the product level builds have been taken off. I told him that's not enough.*
> *LM: Not enough, and really it's starting to hinder our component's development. "* — Developer, project manager and line manager

The second reason for hardware being a bottleneck was that acquiring more of it was a slow process. The case company is a large enterprise, with some inertia in decision making and approval processes. Investments have to be approved where the effects of those investments, or the lack of them, are not directly felt. Even though the teams were constantly underlining the

issue of insufficient hardware, acquiring more would take time.

> *" CI: At some point there was talk of getting more servers, but I don't know what's happened to that.*
> *PM: They have some plan, and some CAPEX planning is being done, but the case is not progressing. "* — CI specialist and project manager

4.4.2 Test Environment Management

Periods of time extending over several days would sometimes be wrought with problems in test environments. The problems would be usually availability or connectivity ones; issues in network setup, user accounts or other configuration. Test environments were maintained off-site, meaning that some problems with them could not be solved right away locally. Even though conceivably some issues could be solved locally through remote access, developers themselves did not always have the competence or confidence to solve them. A ticketing system served as the main communication channel between development and test environment administration. Less official forms of communication were also used, but service tickets acted as a backbone. Ticketing caused some inefficiency in communication, elaborated in more detail in Section 4.2.2.

Planned maintenance breaks in test environments would affect development negatively. The breaks were scheduled to happen over week-ends to minimize impact on development. Regardless of this, their effects would be felt long after, with environments still not responding or configurations being broken even after those weekends. Elongated problems due to planned maintenance were felt several times over the studied time period.

> *" PM: But this is still related to the maintenance problems?*
> *TA: Yeah, the back-end is down. Not up yet. That's why the integration test is red.*
> *PM: Convenient. One day of maintenance, or what, a weekend...*
> *TA: A weekend, yes, it is still... A couple of days is wasted here, that's how it goes. "* — Project manager and test automation engineer

Partially problems encountered after maintenance breaks were traceable to how installations of test environments were handled. Installations would be based on virtual machine snapshots, which would retain some configuration, such as IP addresses. An example of an error caused by this would be, if someone changed an IP address for a test machine, and did not take a new VM snapshot, the change would be overridden when the environment was next reset to the snapshot state. Resets to snapshots and clean installs of

the software would happen each night for most of the components' test environments, as part of nightly extended testing. The purpose of clean installs was both to test the install procedure and to set a clean state for the test environment.

4.4.3 Tool Issues

Some issues with chosen tools were encountered. The studied project had problems with the testing tool Robot Framework. Tests written for Robot were hard to maintain, which led to deprecation of some tests as described in Section 4.1.3. In addition, the build tool, Maven, suffered from some bugs during the study period, causing bad builds and need for re-configuration. Also, a required update in Maven changed the way in which it handled ranged dependencies. As an unexpected result, updating the build tool significantly lengthened some build times.

> *" We've stopped the product level integration build. With the new Maven version it fetches more dependencies. For example, it fetches dependencies from the Maven repository for three hours before it starts to even compile. We will likely have to downgrade the version. "* — Line manager

4.5 CI Practicalities

At the time of the study, the case project is still adapting to its CI process, and faces some challenges in implementing CI practicalities. The project experienced some hardships in *build configuration management*. There was also some trouble keeping up with *frequent integrations*.

4.5.1 Integration Frequency

The teams were not able to always maintain constant continuous integration between dependent components. Changes in components that the teams' work depended on could break builds for the teams. The information value of a build's status for the development team suffers if many of the errors indicated by a broken build are not related to the team's work, but instead are caused by changes further down the dependency chain. The developers would sometimes have to lock the versions of their dependencies to maintain a workable build. This in effect deferred integration between components to

a later time.

> *" DEV: If you match the version to the latest, then we're in the same boat as the rest are. But then, the builds will produce what they will. Their colour will change regardless of our actions.*
> *PM: That's better than constant red.*
> *DEV: Yes, but any commit done can break something somewhere, which has nothing to do with the commit in question. "*
>
> — Developer and project manager

On the other hand, CI literature strongly suggests to always build and test the newest changes, and to integrate all code often to avoid the problems of deferred integrations [e.g. Humble and Farley, 2010, p. 23-24]. The stand-up participants acknowledged that locking dependencies to certain versions, i.e. deferring the integration, was not considered a recommended CI practice by the project personnel. However, it was necessary at times in order to shield the team from interference caused by ongoing work in dependencies.

> *" Of course, when we define older builds (as dependencies), continuous integration isn't exactly practiced. For some (dependency) packages it is practiced, but if there's something suspicious, we want to ensure that we get working ones, so that our own development isn't affected negatively. "*
>
> — Test automation engineer

The teams were aware that deferred integrations may cause more work when finally integration takes place, and did experience this first hand, going through some periods of big, deferred integrations. In practice, a balance needs to be found in how often to integrate code, and this doesn't seem trivial. The studied project was at times suffering from both, too frequent and too infrequent integration. Integrating to latest tested dependencies continuously was a goal, but this goal could not always be met.

> *" LM1: I would link this to merging the code. A few months ago, we were suffering because the builds were constantly red, as our dependencies kept changing. So, to get the team some working peace, we locked the versions, hard-coded some versions into the POM-file, and the builds were fixed with that.*
> *LM2: And now we're enjoying the results of that.*
> *LM1: As we're enjoying the results now, when we switch to the current versions (of dependencies), there's going to be a lot of errors. It's the same with branching and merging. If you merge often, you'll have your build red often and the software broken. If you merge seldom, then you'll have more problems on one go. "*
>
> — Two line managers

4.5.2 Build Configuration Management

Configuration management for CI builds was not trivial. Not all developers were confident enough in their ability to configure the CI pipeline properly, so that sometimes configuration changes would wait for an expert to give support. On the other hand, changes to configuration were not well tracked, and concern was raised about not being able to track changes and revert to a previous state in case of trouble. The project was beginning to adopt measures to track changes in configuration better.

Build job configurations for new jobs were in large part copied from other similar build jobs. This meant that sometimes also configuration errors or some special configurations that were meant for a single job would be copied to other jobs where they would not be appropriate. For example, even definitions for some dependencies to commonly used base components could be affected by copying.

> *" LM: So, your component's dependency should be set up so that it'll fetch the latest green (build), and not just the latest (build) of the dependency?*
> *DEV: I highly recommend this.*
> *PM: I don't really understand why it's the latest one. If we can check that it's the latest green one, then why don't we...*
> *DEV: Yes, it definitely should start up at least.*
> *CI: Configuration issue. Historical reasons.*
> *LM: It's been originally copied from somewhere, where they want the latest (build). "* — Line manager, developer, project manager and CI specialist

Build configuration was so complex that all developers did not have a clear understanding of what exactly happens in the pipeline. Developers do not necessarily have time to learn the internals of CI configuration, while they are at the same time required to produce complex software. While reviewing the results of this study with the participants, one developer reports that this is partly the cause for copying configurations between components.

4.6 Challenges: Summary

In summary, the major challenges observed were related to communication, testing, dependency management, computing infrastructure and CI practicalities, as described in detail in the previous subsections. These high level themes occurred often in the integration stand-up discussions throughout the observation time period of four months.

The issues were highly related to each other. In Figure 4.3, interrelations between these issues are presented as a causal map. The figure gathers to-

gether the challenges that the developers experienced in regards to CI, to better illustrate the numerous connections they have to each other. The nodes in the causal map represent issues previously discussed under the detailed treatise of individual challenges in this Chapter4: observed challenges and their causes and effects. As the issues are based on experiences and expert analysis of the developers, the evidence for the presented causal hypotheses is of a qualitative nature.

The identified causal chains join together to suggest patterns of how problems in the CI process reduce productivity. The causal map converges on the issue of *reduced productivity*. However, in this study, developer productivity was not measured in quantified terms. Thus, the final link to developer productivity remains theoretical. The final identified causes for reduced productivity were identified as *harder debugging* of build failures, developers being blocked by unnecessarily *lengthened issues*, *reduced confidence* of developers in the accuracy of build results and the CI system, and *slow build feedback*. The theoretical link is supported in the literature by other authors who have previously identified the issues issues of harder debugging [Fowler, 2006], low confidence in builds [Neely and Stolt, 2013] and slow feedback [Goodman and Elbaz, 2008; Leppänen et al., 2015; Ebert et al., 2001] to hinder productivity.

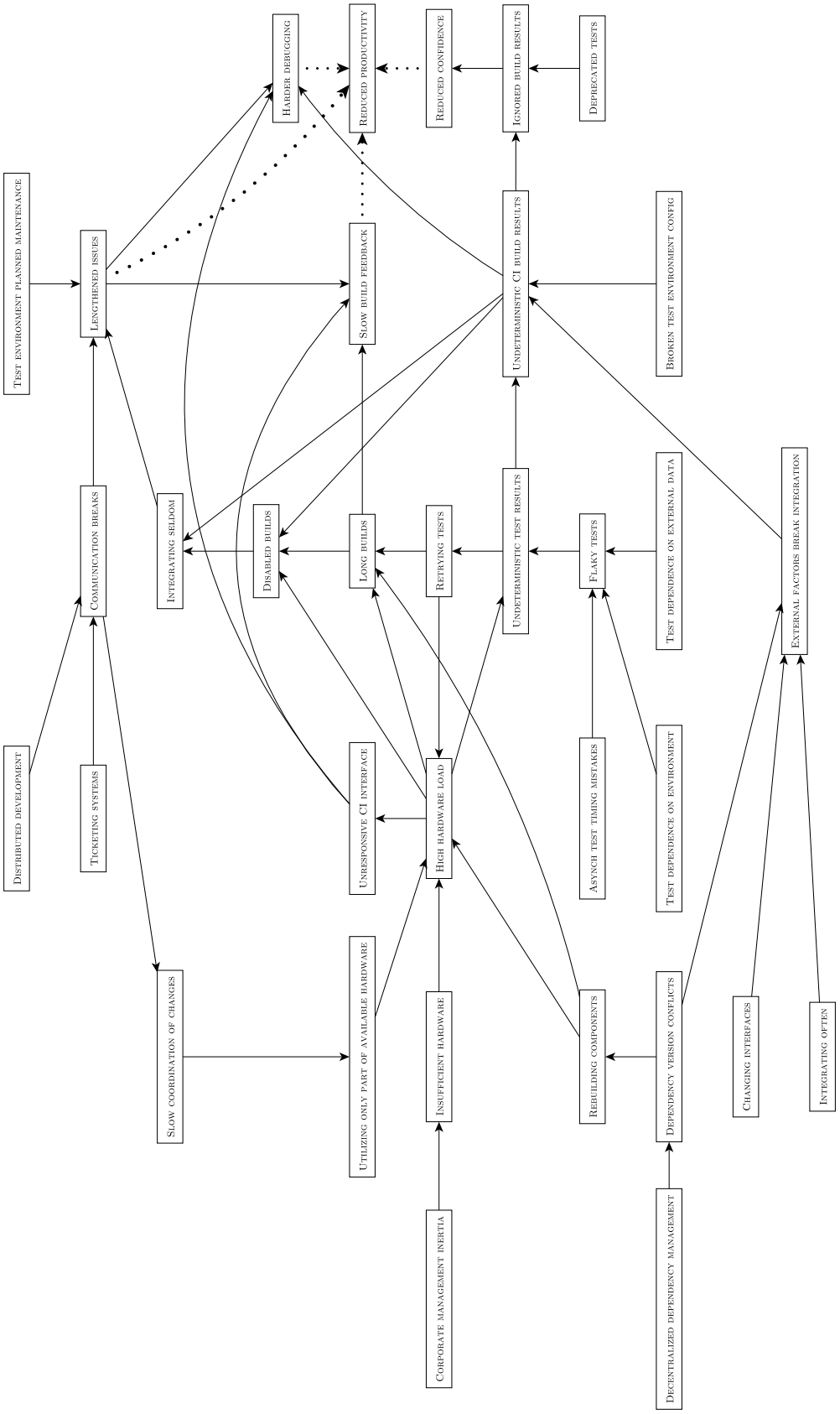


Figure 4.3: Causes and effects of experienced CI challenges, and interrelations between them as a causal map. Nodes represent observed challenges, their causes and effects. Directed edges carry a meaning of causation.

Many of the discussed challenges were related to issues not directly under the team's control in some way. Overloaded hardware, indeterministic tests, changes in dependencies and issues with test environments were often cited to cause trouble. For a lot of these problems, the teams did not have a direct solution at hand. These issues could break a team's build, without the team doing anything, and could take a long time to fix.

The teams followed their component level CI pipelines closely mainly for the purpose of getting feedback on their own ongoing development. When the build was broken by something else than the team's own work, the break could be seen as a hindrance, rather than as informative. Issues external to teams' own work could take longer to solve, and the team might not have the means to solve the issues themselves, perhaps strengthening the feeling of hindrance. As sporadic build breaks were not informative, actions were taken to prevent them. Teams would integrate less often to dependency components to maintain a more consistent build. Sometimes, builds were even disabled in response to sporadic behavior, because their results provided very little desired information.

Chapter 5

Build Failures Quantified

How often did builds fail, and how severe were the failures? Chapter 4 presents qualitative insight to challenges in CI use in the project. How do these experiences relate to data in the CI system? This section takes a look at some of the CI system's data, in order to better answer *RQ4 Can the effects caused by these challenges be quantified by measurements from the CI system?*

5.1 Build Success

Overall, a large number of builds fail for various reasons in the studied project: 69% (20092/28991) of build jobs in the data set were successful. However, this does not tell the whole story. A build job in the CI system of the case project is in most cases a single stage in a pipeline of multiple verification steps. An increment of code is only successfully verified to work after all stages in the pipeline have successfully passed. Thus, success for all stages is a better measure of general build success in this case. 55% (16035/28991) of the builds were triggered by another build as part of a pipeline, leaving 12957 root pipeline invocations. These root builds were triggered either by version control, or manually by users, or by a timer. By tracing success of the root builds through all stages of the pipeline, only 38% (4885/12957) of root pipeline invocations made it through all verification stages successfully.

Figure 5.1 presents the build success ratio over time, averaged over weeks. The solid line represents the success ratio for all builds, while the dotted line below it graphs success of full pipelines, accounting for all verification steps. Full pipeline success follows a pattern similar to plain build success. Drops in success rate are more pronounced for the full pipeline accounting, which is to be expected. When repairing a build that fails at a later stage of the pipeline, a lot of successful builds are automatically generated by the CI system at earlier stages of the pipeline. Thus, only looking at the success rate of all builds would belittle the occurrence of build failures in the pipeline.

Two disturbances in the project were identified that closely coincide with clear drops in CI build success rate. Twice during the studied period, build success rate reaches a low point of only approximately 25% of initiated CI builds passing through the whole pipeline without failure. Both periods of low success are preceded by a sudden drop in success rate. The originating weeks of the identified disturbances are marked on the time axis of Figure 5.1 as the vertical lines labeled *p1* and *p2*.

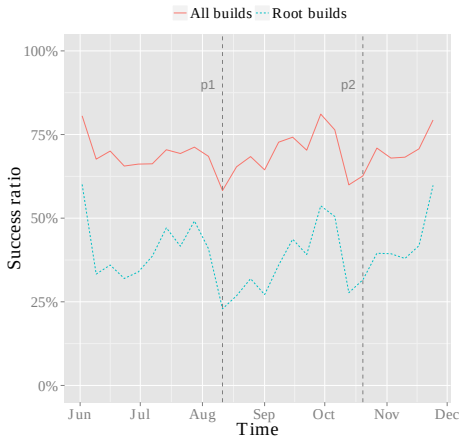


Figure 5.1: Weekly mean build success rate over time. The labels *p1* and *p2* denote identified disturbances in the software development process.

One of the disturbances can be explained by the stand-up observations. The observation data of integration stand-ups reveals that in mid-August, on the week labeled *p1* in the figure, test environments underwent some planned maintenance. Build jobs relying on the test environments that were attempted during the maintenance break would necessarily break. However, even after the break was over, a lot of build jobs would not recover. The break left many test systems in a mis-configured state, and the effects were

felt long after that. Developers would have to spend time waiting for the test environments to function. The test environments were physically located and administered off-site. Developers did not always have the means or expertise to fix them themselves.

Another disturbance, labeled *p2* in the figure, took place in October. The stand-up observations do not cover this period of time to give detailed and timely insight to problems experienced during the week *p2*. In an additional e-mail interview, a line manager reveals that there were changes in the project during that week. The disturbance *p2* can be attributed to a re-organization of development teams. Responsibilities for software components were re-arranged among teams, and the team composition between different sites changed to a degree. This involved a number of established developers in the project being assigned to another project. According to the interviewed line manager, this led to a small period of unclarity in the project.

5.2 Speed of Repairing Broken Builds

Measuring the time it takes to repair a broken build has been used by several authors to measure CI processes [e.g. Miller, 2008; Downs et al., 2012; Kerzazi et al., 2014]. Miller [2008] and Kerzazi et al. [2014] both use time to repair to estimate the costs of time lost in repairing builds. Downs et al. [2012] show that developers' increased awareness of build status reduces the time it takes to fix builds. Practitioner literature also stresses the importance of fixing builds fast [Fowler, 2006; Humble and Farley, 2010, p. 67-71]. Thus, time to repair a broken build can be indicative of CI process maturity in many ways. The median is chosen as the representative measure of fix time, because it is robust against the effect of abnormal outlier values. The stand-up discussions reveal that there were a number of occasions where the developers ignored certain failed builds for extended periods of time, mostly because of deprecated or highly unstable tests. In the data set, the longest time it took for a build to be repaired is 50 days. For this reason, the median instead of a mean value is expected to give a better estimate of how fast the teams generally fixed broken builds.

A time series of the weekly median build fix time over the duration of the study is presented in Figure 5.2. Because of large single variations in median fix time, this indicator does not have a clear trend over time. It is more indicative of widely felt problems that affect build repair time for the whole project. The median fix time for each week is calculated according to the fix time of all builds that were broken on that specific week. Fix time is accounted for the week of breakage regardless of how long the build took to

fix. Thus, problems that widely caused lengthened repair times converge on the graph as an increase in median fix time on the week of the origin of the problem.

As discussed earlier in Section 5.1, two larger disturbances to the software development process were identified during the course of this study. These disturbances are clearly evident in the median build repair time metric shown in Figure 5.2. The disturbances were related to lengthened test environment maintenance in August (*p1*), and a re-organization of development teams in October (*p2*). During the weeks of disturbance in both October and August, fix times of generally two hours increase to around ten hours for builds that failed on those weeks. At the same time, the success rate of builds completing all stages of the CI pipeline drops from around 50% to 25%, as displayed in Figure 5.1. This shows that not only were builds more often broken during those weeks, but also took a significantly longer time to fix. Builds were attempted, but were often broken, and took a long time to fix.

The median CI build fix time is a strong indicator of problems in a software project. In the studied case, clear deviations in weekly median fix time provided much stronger indications of widely felt problems than build success rates.

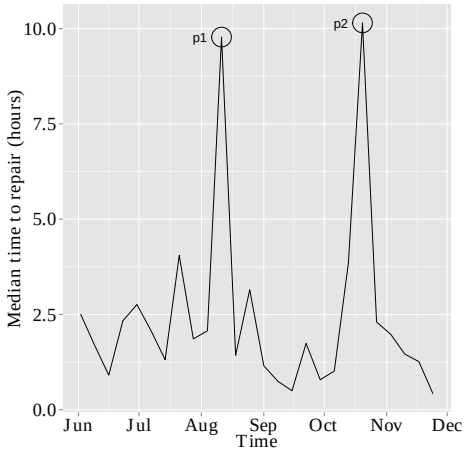


Figure 5.2: Weekly median time to repair broken builds. Identified wider disturbances in the software development process, labeled *p1* and *p2* are clearly evident from median repair time.

Another measure that could also be indicative of the rigorousness of CI practices is the number of builds that it takes for the developers to fix a broken build. A hypothesis would be, that as the CI practice is more understood

in an organization, the number of builds required to fix a broken build would decrease. As is noted in the CI literature, to ease debugging, any code that is not related to fixing a broken build build should not be committed when the build is broken [Humble and Farley, 2010, p. 66-68]. A low number of attempted builds required to fix a break may tell of a better focus on the fix.

A downward trend in the number of builds required to fix a broken build is apparent throughout the studied time period, although this number exhibits considerable variance. Depicted in Figure 5.3 is a time series of the weekly mean number of builds to repair a broken build. The value for each week is the mean among builds that were broken on that week. The downward trend in this number may tell of increasing effort by developers to fix a broken build without committing other new code. The trend may also tell of a reduction in long standing issues. However, strong conclusions about process improvement can not be made based on this number. When presented with these measurements, the participants of the study are cautious to attribute the apparent trend to a single improvement. Developing software is a complex process. The decrease in the number of builds it took to repair a failed build could also be attributed to other factors. One example of such a factor, as suggested by the participants, could be increasing maturity in the program code of developed components.

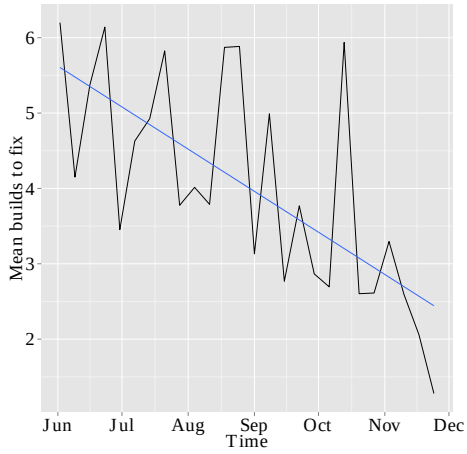


Figure 5.3: Number of builds to fix a broken build. Weekly mean values.

5.3 Measurable Effects of Different Failure Causes

To gain an understanding of which failure causes were most commonly behind build failures, and to investigate the effects of different kinds of failures, the developers took time to log their analysis of build failure root causes into a Jenkins plug-in, as described in Section 3.4. By logging the categorizations into the CI system, other CI data could be directly related to the categorizations. Table 5.1 outlines the categorized data. It provides a brief description of the categorization system used by the developers, as well as the occurrence and median repair times of build failures categorized under each category.

Category	**Interpretation**	**Occurrence (n=126)**	**Median time to repair**
CI	CI environment and configuration problems.	23 (18.3%)	14.09 h
CODE	Programming mistakes.	37 (29.4%)	1.27 h
INSTALL	Problems with automatic deployment scripts.	8 (6.3%)	9.74 h
TEST	Test implementation mistakes.	12 (9.5%)	0.53 h
TEST-DATA	External test data missing or misconfigured.	7 (5.6%)	0.96 h
TEST-ENV	Test environment issues.	13 (10.3%)	16.96 h
TEST-UNSTABLE	Unstable or flaky tests.	9 (7.1%)	10.09 h
OTHER	Other failure causes.	14 (11.1%)	1.02 h
VERSIONS	Dependency version mismatches.	3 (2.4%)	4.27 h

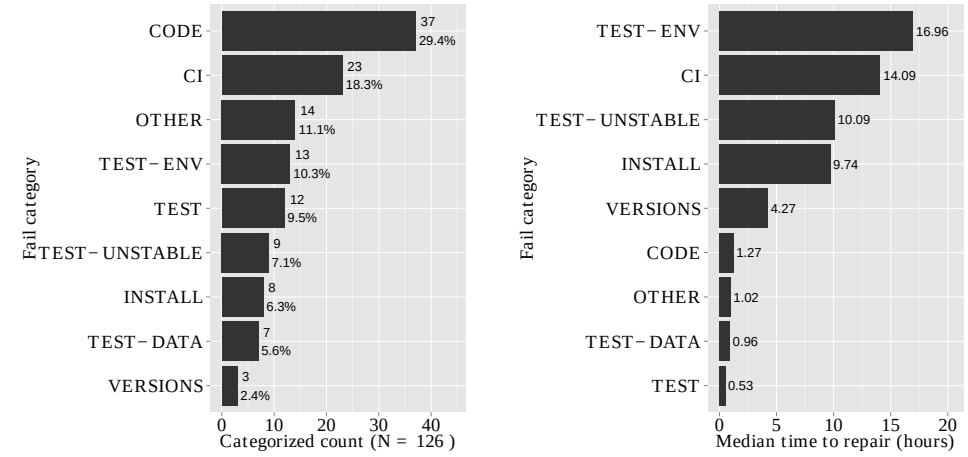
Table 5.1: Categorized build failures.

The categories have been combined from more detailed ones used by the developers. The developers logged test environment failures for two subsystems of the application under different categories, whereas the analysis combines these into one category, *TEST-ENV*. Similarly, the category *INSTALL* also combines detailed categories for different subsystems. Finally, the original categorization had separate categories for configuration mistakes specifically, as opposed to hardware issues or others. Configuration categories existed for both the Jenkins CI system and the test environments. However, by analyzing additional notes in developers' categorizations, it seems that the configuration categories were mostly used interchangeably with the corresponding categories reserved for other issues. It is thus hard to justify a separate analysis for configuration categories. As a result, the categories *CI* and *TEST-ENV* include all kinds of failures attributed to those systems.

A total of 126 builds that broke a line of successful builds were categorized. To analyze the possible effects of different kinds of build breaks, median repair times for each of the categories were calculated from CI data. Repair time is measured from the end of a broken build to the end of a build that finally fixed the error. End time is chosen as the point of measure, because feedback

to the developer is possible only when a build finishes; a build is proven to break or fix the build status only when it's run its course. During the time that a build was broken, several broken builds could be produced, and the first non-broken build was considered to fix the build.

In Figure 5.4, the differences in (a) occurrence and (b) median repair time of categorized build failures are visualized.



(a) (b)

Figure 5.4: A different view on the categorized build failures. (a) Number of build failures categorized for each category. (b) Median time to repair by failure category.

Most of the build breaks were categorized as issues in program code by the developers. 29% (37/126) of the breaks were categorized as code issues. The *CODE* category included such errors as programming mistakes, uncommitted files, and bad definitions for dependencies. The teams did not consider this category of broken builds to be problematic. CI is supposed to bring light to human mistakes, and this category represented exactly those: mistakes made in development. In their high level root cause analysis, the teams did not differentiate between errors caused by the team themselves and those introduced by problems in dependencies, perhaps developed at other sites. A coding error in a dependency would be categorized into this category, although the team themselves perhaps could not solve it. Code problems had relatively fast fixes applied to them, with a median build fix time of slightly over one hour.

Problems in the CI environment, categorized under the category *CI*, were the second most often logged build failure category. These would include

erroneous configuration of the CI system or that of a build job, bugs in the Jenkins CI system or its extension plug-ins, and breaks caused by sporadic behavior of the CI machine or software infrastructure. 18% (23/126) of categorized build breaks were accounted to issues in the CI system. These build breaks would take considerably longer to fix, with a mean fix time of 14 hours. More often than not, fixing would wait until the next business day. In fixing issues in CI configuration, the teams often needed support from specialists. Added communication and coordination delays thus partly explain the longer time to fix problems in the CI environment.

Testing problems were abundant and generally took some time to fix. Test implementation issues and mistakes in updating tests as functionality changed were reported under the *TEST* category (9.5%). Issues in test environments, such as connectivity problems or configuration ones, were assigned to the *TEST-ENV* (10.3%) category. Instability in tests as a root cause had its own category, *TEST-UNSTABLE* (7.1%), as did problems in handling external test data, *TEST-DATA* (5.6%). Build breaks accountable to test environments had the longest median fix time of almost 17 hours. The prolonged fix time of test environment problems mirrors the challenges reported by the integration stand-up participants well. As the environments were maintained at a remote site, communication and fixes related to them were not as efficient. The communication problems brought forth in the stand-up discussions are detailed in Section 4.2, and were often related to test environments. As seen in the integration stand-ups and described in Section 4.1.1, there was reduced confidence in builds that had known unstable tests. These tests were hard to debug because of their indeterministic nature, and thus harder to fix. Sometimes a failed build result that was assumed to originate from flaky tests could even get ignored for a time. Therefore, a longer median fix time of 10 hours for the *TEST-UNSTABLE* category is not surprising.

Installation issues, under the category *INSTALL*, were problems in automatic deployment of software into test environments. Whereas issues in actual test servers were logged under the *TEST-ENV* category, the failures under the *INSTALL* category represent scripting problems in automated deployment. Examples include missing or conflicting installed software and problems in the order of automated start-up of software or environments. 6.3% (8/126) of categorized breaks were labeled as such. The median fix time was about 10 hours. Few specialists had deep knowledge of these installation scripts, which is one reason for their prolonged repair time.

The *VERSIONS* category was set up specifically to track version mismatches in dependencies, resulting from multiple packages defining the same dependency with slightly different versions. The developers had recognized this as a reoccurring problem, as previously discussed in Section 4.3.2. How-

ever, the developers categorized only 3/126 (2.4%) of the analyzed build breaks to be caused by this reason with a comparatively reasonable median fix time of about 4 hours.

The *OTHER* category contains build breaks that did not fit any other category. 14/126 (11.1%) were categorized as such. The entries in the *OTHER* category, according to additional notes entered by the developers, include some build breaks that were logged during a large refactoring operation, some specific software configuration issues and some seemingly sporadic failures. The breaks in this category generally received quick fixes.

Concluding from the big differences in repair time between categories, and reflecting against the knowledge gathered in the stand-up sessions, the evidence supports that the fastest issues to fix were those that the teams had direct control over. The fastest measured breaks to fix fell within the categories of code and test implementations, problems in test data and version mismatches. Breaks caused by test environments, the CI system and installation scripts all would often require support from specialists or collaboration over site boundaries to fix, adding to the time required to repair them.

5.4 Build Breaks in the Pipeline

In the studied project, a single CI build in most cases does not represent a complete compilation and testing cycle for a code increment. Instead, the verification pipeline is split over several CI build jobs. The standard CI setup for any component in the project has multiple stages as separate CI build jobs, as described in Section 3.3.1. It includes four stages: build, install, integration and acceptance tests as separate CI jobs. In addition, nightly automated operations have been configured as separate stages titled scratch and slow. The distribution of build breakages across these pipeline stages can be seen in Figure 5.5. This graph includes all failures for the project in relevant build jobs over the studied half-year time period.

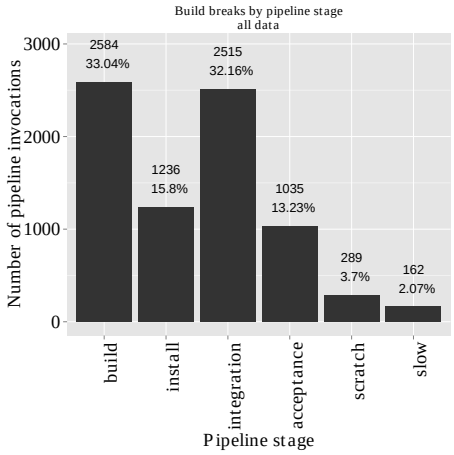


Figure 5.5: Build failure distribution over different stages of the build pipeline.

Approximately one third of the build failures have occurred in the build stage. Another third of build breaks have taken place in integration testing, with the remaining third being split somewhat evenly among the stages of deployment to test environments and acceptance testing.

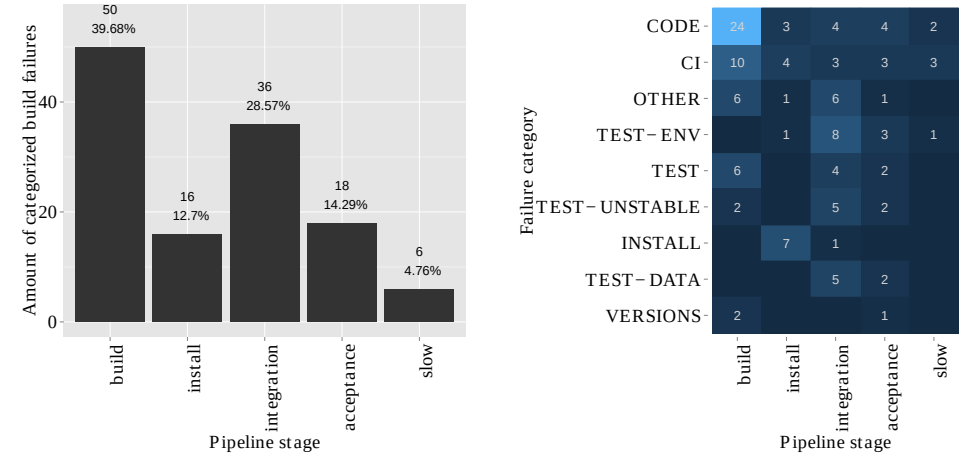
The main purpose of splitting the verification cycle to smaller stages is to hasten feedback of possible errors, while still maintaining sufficient quality assurance. By investigating the number of failures, indeed it would seem that a large number of them happen in the earliest, build stage of the pipeline, providing fast feedback. In this regard, the multi-stage build pipeline is proven valuable.

A large number of failures in the integration test stage may be accountable to the fact that some integration tests in the case software product can only be run in the test environments. The developers are unable to run these tests on their local development machines before committing the code for the CI system to verify. Even if they could be run from local development machines, there are differences between the development and testing environments that can cause differing test results. Furthermore, in the stand-up discussions, problems with test environment configuration, test dependence on specific data sets and deficiencies in test implementation were common topics. The integration test stage is the first stage of testing performed in the test environment that can reveal all of these problems. Therefore, there is a multitude of causes each contributing a small portion of the large number of failures in the integration test stage.

The nightly operations, labeled as pipeline stages scratch and slow, un-

derstandably contribute less build failures, as they are run less often than builds triggered by version control. The failures in the scratch stage are of less importance to the developers' daily information flow. The scratch stage sets a test environment to a predefined clean state, and thus does not tell much of the state of the software itself. Slow test routines include the operations of all the daily stages combined, and may include additional testing that is deemed too slow to keep in the daily routine. Thus, the nightly slow test routine is likely to succeed if the daily stages do so, but may also reveal mistakes that the daily cycles do not.

To study the implications of the distribution of failures over build stages closer, Figure 5.6 relates the developers' categorizations of failure root causes to the pipeline stages where failures occurred. Figure 5.6 (a) shows that the categorized builds have a notably similar distribution over pipeline stages compared to the whole dataset, in Figure 5.5. Figure 5.6 (b) then relates the categorizations to different pipeline stages.



(a) (b)

Figure 5.6: Failure causes and pipeline stages. (a) Build failure distribution of categorized builds over pipeline stages. (b) Categorized failures by pipeline stage.

The similarity of the two distributions of failures across build stages apparent in Figures 5.5 and 5.6 (a) raises hopes that the developers possibly did not favor analyzing certain kinds of failures over others. However, this similarity is not statistically significant. Disregarding from the analysis the measurements for the scratch stage of the pipeline, a Pearson chi-squared test for goodness of fit [Cochran, 1952] between the two distributions fails to confirm their statistical similarity (p-value 0.15). The failures of the scratch stage are overlooked in this analysis. The reasoning for the omission is that the because of the janitorial nature of the scratch CI stage it is less important to the developers. There were no recorded developer failure cause analyses for the scratch stage. In the reference distribution of all builds, the proportion of failures in the scratch stage is also quite small. Thus, including these failures in the calculation would misguidedly offer a better match between the two distributions. Some bias of developers may still be present in selecting which kinds of failed builds to analyze and categorize.

As can be seen in the break-down of categorized build breaks in Figure 5.6 (b), a majority of coding errors and a large number of problems in the CI system or its configuration are caught at the earliest, *build* stage. This allows for fast failure feedback, and indeed coding errors are also fixed relatively fast, with a median fix time of 1.27 hours. However, even though CI problems are in large part found already in the build stage, they took a much longer time to fix. This may be because of the causes of failure being unclear to or out of control of the developers. Judging by the additional information recorded by the developers at the time of categorization, these CI errors would generally be experienced as sporadic, or to be caused by interfering changes from outside the team. Although the speed of feedback is important for the CI process, it does not necessarily dictate fix time. Bottlenecks exist elsewhere that have a large effect on fix time, and can overshadow the build feedback time in magnitude. These bottlenecks may be various, such as described in Chapter 4. The quality of feedback is also important: if the results of the build are not trusted, there may be lowered incentive to try to fix it.

Builds that failed in the integration test stage exhibit a wide variety of categorized root causes for failure. Where a large majority of the failures caught in the build stage were due to human errors in coding or testing, in the integration test stage human errors are a minority. The types of failures in the acceptance test stage are distributed similarly with the integration test stage over the failure categories. Acceptance tests in the studied project's CI pipeline are also performed in the same dedicated test environments. In the integration and acceptance stages, a lot of work is expended on fixing build failures that do not provide any information for the verification of code

increments. Availability and configuration of environments, lacking test data and unstable tests act as exemplary causes of such failures. Thus, maintaining the complicated test environment setup is expensive to the project, while providing diminishing return measured in revealed human errors compared to earlier and simpler test stages. Regardless, the testing environments are essential to the project. Testing against a system that is similar to a production environment provides more accurate results.

Chapter 6

Discussion

This chapter concentrates on internal and external reflection of the research. Section 6.1 discusses how the findings of this study fit into previous literature. Section 6.2 reflects upon the research methods used and their possible shortcomings. And finally, some thought is given to a specific data collection method used in this study: Section 6.3 discusses the utility of manually categorizing CI build failures.

6.1 Reflection Against Literature

The brief treatise on CI problems in literature, in Section 2.5, reveals similar findings between this study and previous ones. This section reflects learnings from literature to the particular circumstances of this case, hopefully providing some grounds for wider generalization of some issues. Relating the major challenges that the case organization faces in its CI use to literature, I find that most of the challenges apparent in this case have been also previously identified as challenges in a CI context. The following subsections relate major challenge themes identified in the context of this case to earlier literature.

6.1.1 Testing

Automated testing is a major concern for implementing CI, noted in literature and observed also in this very case. Looking at the durations of build fixes reported in Section 5.3, it appears that unstable tests and the difficulties of maintaining external testing environments caused lengthy build breaks in the case project. The project had some difficulties in test implementations

also, especially with the timing of asynchronous testing and tests being dependent on environments or external data.

Unstable tests have been reported to lower the confidence of developers in build results. [Neely and Stolt, 2013] This effect is clearly present in the studied project as well. As described in Sections 4.1.1 and 4.1.3, some unstable tests in the product could go ignored in the studied project for long periods of time. A few unstable tests reduce the value of the whole test set. The benefit of accurate and timely information about newly created bugs in the code is lost. Also, with unstable tests in the test set, developers can lose confidence to do major changes backed by a working test set that would catch problems as they are created. [Fowler, 2011]

Implementation of asynchronous testing was one area where the case project faced challenges. As seen in Section 4.1.2, especially the intricacies of the timing of verification and the changing DOM structure were discussed in stand-ups at length. These were related to unstable tests as well. Mistakes in test implementation regarding these details could result in flaky tests. Both the DOM update problem and the timing one have previously been noted in literature as problem areas regarding asynchronous testing. Marchetto et al. [2008] present these same problems as typical faults in AJAX applications. Luo et al. [2014] classify verification timing errors as *async wait* errors, and note that they are a common cause of flaky test implementations. In an analysis of VCS commits fixing flaky tests, they found 45% of studied flaky tests to fall under this category.

Of the unstable tests, some were considered too hard to maintain, and as such deprecated. As described in Section 4.1.3, no effort was expended to maintain some tests. It may be notable that most discussion around deprecated tests involved tests written for Robot Framework. There are conflicting reports in literature about the maintainability of automated tests created with Robot Framework. Emery [2009] demonstrates how a keyword-driven testing framework, such as Robot Framework can increase the maintainability of tests by hiding incidental details, removing duplication and by increasing the readability of tests. In contrast, Rantanen [2010, p. 44-45] observed in a practical industry setting that the inherent flexibility of Robot Framework can cause a lot of variability in how different people implement tests. The framework provides very little structure of its own for tests, leaving everything up to the individual developing the test. This in turn makes maintaining a collection of tests written by multiple people harder.

As can be seen, the implementation and technicalities of testing are widely relevant challenges for the case project. They also generalize against other cases quite well. This study affirms the previous belief that automated testing is a major CI cornerstone, further explaining some reasons behind this

statement.

6.1.2 Management Support

Heavily cited both as a success factor and a challenge for CI adaption processes is management support [Neely and Stolt, 2013; Leppänen et al., 2015; Debbiche et al., 2014; Claps et al., 2015; Olsson et al., 2012]. Because the challenge is so prominent in literature, it should be discussed briefly.

The main observational data collection method of this study does not lend itself well to studying the effects of management support on CI adaption. The focus of this study is more closely on challenges that are visible to the developers in their daily work. However, as described in Section 4.4.1, one adverse effect regarding management inertia was indeed experienced by the case personnel. Acquisition of needed additional servers for running the CI pipeline was delayed because of a lengthy corporate decision making process. This does not directly replicate issues reported in previous literature, but does underline how corporate structures can affect even daily development work. Thus, I can support the notion of previous research, that future studies should stay alert about the possible effects of management support on CI adaption.

6.1.3 Communication and Distributed Development

Distributed development is a known cause of delays and inefficiencies in communication, as written in previous literature [e.g. Herbsleb and Grinter, 1999; Herbsleb and Mockus, 2003]. Section 4.2.1 demonstrates that the case project exhibits trouble synchronizing development between sites. Communication gaps were especially visible in off-site test environment management, described in Section 4.4.2. As can be seen from Section 4.2.2, ticketing systems as a communication channel also played a role in distancing communication.

These communication drawbacks fit well to those described by Herbsleb and Grinter [1999]. In a distributed environment, people do not as readily and actively contact others for help. There is an increased cost of initiating contact. Getting help across site borders is more tedious: there is less visibility on who is available, and uncertainty about who exactly can help with each specific problem. Time differences may further add to the difficulty of contacting the right people. Co-located development has the benefit of frequent unplanned contact. Because of this, tacit information and news are shared somewhat efficiently within a site, but sharing does not occur as easily over site borders. Also, as communication is far removed, responsiveness is reduced. Respondents to e-mails may have difficulty in evaluating

the importance of inquiries, and response times generally grow. It is harder to see and experience first hand the effects of problems, reducing the feeling of urgency. [Herbsleb and Grinter, 1999] These inefficiencies are very general in nature. They would be most likely relevant to any project of substantial size. The CI context presents additional flavors to these problems, so that the communication inefficiencies make the organization and efficient use of CI harder.

6.1.4 Infrastructure

Insufficient hardware, test environment management and bugs in tools were among the more commonly faced problems in this study. Of these, there is ample evidence that environment management had major effects on the software integration process. As seen in Section 5.2, issues in the case's test environments could dramatically lengthen build failure times. The analysis of repair times for different failure causes in Section 5.3 further supports this conclusion. Additional analysis in Section 5.4 reveals that despite the large disturbances caused by test environments, few human programming errors were caught by them, whereas the environments themselves were to blame for at least as many errors as they revealed in the program code.

Why were problems in environment management so prominent? Developers' lack of confidence or access in fixing environment related problems tied their hands from efficiently solving issues themselves. Communication gaps between development and an off-site maintenance team was another reason for delays in fixing these problems. Thus, in fact, the environment management issue is caused by problems in communication, distributed development and a lack of expertise or access of developers. Problems with environments are how these issues are apparent to the developers, but in fact the reasons lie deeper. The fact that environment management issues appear pronounced in data may partly be a result of the chosen research design. The observational method and reliance on expert analyses of failure causes both reveal problems as they practically appear. Fortunately, the richness of qualitative data in this case allowed for making deeper inferences.

Tool maturity is a common grief for CI use in the literature [e.g. Olsson et al., 2012; Debbiche et al., 2014]. This seems to hold true to an extent for the studied case as well. Automated test and build tools were found lacking, as reported in Section 4.4.3. The choice of tools for a project can have long lasting and expensive effects.

6.2 Threats to Validity

This section details possible methodological shortcomings of this study and reports on actions taken to control them. Yin [1994] divides validity threats to two high level categories: *internal* and *external* validity.

Internal validity refers to the extent of how well the results of the study represent reality. The researcher's interpretation or different kinds of bias in subjects can prove a threat to truthful representation of circumstances. The researcher himself may also have vested interests or bias for producing desired kinds of results. And finally, the chosen research methods may have some weaknesses in and of themselves. Controlling internal validity is extremely important for this particular case study, as the analysis performed here has been in large part qualitative. Internal validity will be discussed in sections 6.2.1 and 6.2.2.

The notion of external validity deals with how well the results of the study are generalizable to other contexts, and if the results are useful to a wider audience. This study is a single case study performed in one project in an industrial setting. It is therefore clear that external validity needs some thought. The treatise of external validity is covered below in Section 6.3.1.

6.2.1 Observation and Interpretation

A researcher's understanding of the case being studied is not perfect. No amount of data collection can be guaranteed to reveal every single meaningful detail, and necessarily some information is missed. Tacit knowledge exists in the studied organization and project, which may be relevant to the study but very hard to elicit out of the organization.

To understand the studied case in depth, we have performed preliminary interviews concentrating on the case and its CI use before beginning the research. We have also familiarized ourselves with the company's internal CI process documentation. The research question *RQ1 How does the studied project use CI?* was intentionally set to describe the case context in detail. Moreover, the data collection method of continuously observing the case personnel required us to be present at the case site daily for the duration of four months. Additional discussions were held with case personnel during these months to clarify understanding where it was lacking. Even though these measures were taken, it is clear that our understanding of the case context remains worse than that of people who have worked in the project for years.

Any analysis, whether it is qualitative or quantitative in nature, is always

an interpretation made by the researcher of the real world. The meanings and conclusions drawn from data are limited by the researcher's ability to reflect to wider knowledge. Also the researcher may have intentional or subconscious bias toward producing certain results.

I admit that there are obvious limits to my personal knowledge. This study has had the benefit of frequent peer reflections by my colleagues, who are researchers in the field of software engineering and processes. Having their input has been extremely helpful in triangulating methodological and interpretative errors.

To provide transparency into possible vested interests, it should be noted that this research has been funded as a part of the *Need for Speed (N4S)* [1] research program, mostly publicly funded by the Finnish Funding Agency for Innovation, Tekes. The research program has defined a high level objective of studying the improvement of *real-time value delivery*. This study on CI adoption challenges provides a small contribution towards that goal. The program or its administration did not limit or directly guide the research.

6.2.2 Data Collection and Methods

The main data collection methods of observation in situ and relying on data within the actual CI system are not very susceptible to bias. Compared to for example some interview or survey methods, there was no need to design exact interview questions, which could possibly direct the participants to specific answers. The situation was observed as is, and themes in the data corpus are more emergent than guided. There was no need to choose participants for this research, as an existing practice was observed. The results could not be then affected by deliberate choice of certain participants. The results of the study have been presented to the participants, who were content with their accuracy. In these regards, these methods of data collection suffer less from structural guidance.

The continuous observational method chosen allows for a good understanding of the context of the case, and can reveal issues that participants could avoid in interviews [Patton, 2002, p. 261-264]. The main advantage of the observational method is also its main disadvantage. Discussion can not be as easily directed towards relevant issues. For this reason, some interesting aspects of CI adoption were not covered by this study. The effects of management, corporate decision making and the release process on CI are examples of such interesting issues that could not be covered in depth.

[1] Need for Speed (N4S) research project website.
`http://www.digile.fi/N4S`

On the other hand, no study can answer all questions, and a more focused approach allows for deep analysis and concise presentation.

One subset of data remains suspect to some bias. Some data to categorize failed CI builds was gathered during this study, by letting the developers log their analysis of failures into the CI system. Even though categorizing a build did not take much effort, the ratio of categorized build breaks to all breaks remained very low. Thus, the set of categorized breaks may not accurately represent the set of all broken builds. Developers may have chosen to categorize certain failures more eagerly than others. This is a shortcoming of this study, and could have been improved. Minimal disruption to the normal development process was desired for this research. Thus, the developers were encouraged, but not required to log the failure categorizations into the system.

6.3 Usefulness of Failure Categorization

In this study, manual failure categorization did not prove very valuable to the developers themselves. This combined with a desire to minimally disturb the development process in studying it led to a relatively small number of recorded categorizations. A better intrinsic motivation for categorizing may be needed, such as fast and useful feedback based on the categorizations.

As is, the categorization system provides little immediate value to the developers. It allows them to record additional information in a free text field, which was often used for describing the cause of build failure, or for proposing solutions, possibly sharing information between developers. Categorizations for multiple builds were visible when listing a history of builds of a component, giving a quick understanding of recent occurrence of different types of failures. As these features were not incentive enough for categorization, other kinds of useful, immediate feedback should be studied to encourage categorization of more builds. The information value and usability of the current categorization plug-in implementation could also be brought to question.

A more automated approach to categorization could yield a more significant number of categorized builds. The *Build Failure Analyzer* plug-in [2] for the Jenkins CI system is an example of an automated method to achieve failure categorization based on examining build log files. In addition to just categorizing failures, it provides instant valuable feedback to the developers by pointing out detected problems from log files. This potentially helps the developer in debugging build failures. As of writing, it has been installed in

[2] Jenkins Build Failure Analyzer plug-in.
`https://wiki.jenkins-ci.org/display/JENKINS/Build+Failure+Analyzer`

more than 2000 Jenkins systems, and the number is growing. The popularity may indicate that at least automatic categorization can be useful to projects.

An interesting question remains how accurate automatic and manual categorization methods are in describing failure causes, compared to each other. The automatic method has the benefit of being more deterministic and traceable. On the other hand, by analyzing build failures by hand, the developers can extend their analysis to deeper issues that are not necessarily evident straight from a log. These issues could be difficult for an automated method to catch.

Even though the categorizations were not very useful to the developers in their daily work, to us researchers they provided interesting data. Joined together with other CI data, they could help us understand which build failures generally cause more disruption to the development process. These results are reported in Section 5.3. Thus, failure categorization could be a tool for projects using CI to understand and improve their software integration process. The ways in which failure categorization is best implemented and how its data is utilized to best support the developers in their work should be studied further.

6.3.1 Generalizability

The research performed is a single case study, done in a single context. As such, it is mostly limited to explaining the circumstances of the particular studied case.

Even though this is a single case study with limited possibility to generalize, the findings of this study can be reflected against the existing body of literature to confirm or disprove previous findings. For the most part, the challenges in CI use reported herein are not unique to this study, as discussed in Section 6.1. The results of this study thus provide one piece of further evidence for the prevalence of certain CI challenges in an empirical industry setting.

Chapter 7

Conclusions

To close this research report, final concluding remarks are gathered into this Chapter. The results of the research are condensed into a brief summary in Section 7.1. Possible scientific and managerial contributions of the study are discussed in Section 7.2. And finally, Section 7.3 gives pointers for future research based on the results and experiences of this study.

7.1 Summary of Results

This study has concentrated on finding challenges in adopting CI. Its results can be best summarized by looking again at the research questions.

RQ1 How does the studied project use CI? To give context to the case, a detailed account of CI use in the case project has been given in Section 3.3. The project is large and its personnel geographically distributed. Thus, its integration and testing effort is complex, all orchestrated as build jobs in the project's Jenkins CI system.

RQ2 What are the challenges to efficient CI use in the project? The problems that the case project faces in CI use have been studied in depth by performing a thematic analysis of observed integration stand-up meetings. As described in detail in Chapter 4, the case project experiences several challenges in using CI effectively. The problems they face can be summarized to fall under the general level themes of *testing*, *infrastructure*, *dependency management*, *communication* and *CI practicalities*.

RQ3 How do the challenges affect development? The challenges, their causes and effects are highly interrelated and interfere with each other. Based on the stand-up discussions, the mechanisms by which these challenges affect software integration and each other have been mapped as a causal diagram in Figure 4.3. Various causal chains are identified that reduce the productivity

of software development. Ultimately, *slow build feedback*, reduced confidence in build results and the CI system and difficult debugging seem to be major factors in reducing productivity.

Among the more interesting findings of the causal analysis, this case challenges the benefits of very frequent integrations. Developers could not always achieve constant integration, as ongoing development in a large number of dependencies could cause constant failures of dependent builds. Less frequent integration with dependencies could allow the developers to work in peace. The interference caused by dependencies may be a result of specific change management, software architecture and dependency management choices in this project. However, these facets of a software project are difficult and slow to change, and in the mean while, the project still needs to produce software. The case product organization is in a transitional phase, where they aim for continuous integration, but are not yet able to achieve it fully. Integrating every increment of code instantly may not be a desired state for all projects.

RQ4 Can the effects caused by these challenges be quantified by measurements from the CI system? The measurable effects of identified problems are studied by relating qualitative understanding to data from the CI system. In this effort, a categorization of failure causes for individual builds was instrumental. Chapter 5 presents CI system data, relating the data to earlier learnings of thematic analysis.

By combining data in this way, it can be concluded that a the complicated test environment setup in the studied project causes significant additional costs. Test environment issues take a long time to fix due to the specialized expertise required to fix them, and because the environments are maintained off site. Most human errors in program code and test implementations are caught before testing in these environments, while the test systems cause some additional problems of their own. Regardless of this, the environments are crucial to achieving comprehensive testing in the project.

Another conclusion that deserves emphasis is, that whenever developers did not have direct control or competence for solving issues, build breaks could be prolonged. From comparing the fix times for different failure categories, shown in Figure 5.4 (b), it is apparent that problems in test environments, the CI system, and unstable tests took the most time to receive fixes. Limited access to test environments and insufficient confidence of the developers in dealing with configurations were cited as reasons behind this.

7.2 Contribution

The results of this case contribute to affirming previously identified challenges in CI use. Additionally, this study makes an attempt at providing some explanations to the complex mechanisms behind challenges in CI use.

Most case studies about CI or CD use utilize an interview method. Not many previous studies have set out to combine qualitative and quantitative methods to explain the underlying problems in CI processes. Kerzazi et al. [2014], Ståhl and Bosch [2013] and Miller [2008] have done exemplary work in combining qualitative and quantitative data. Mixed methods are particularly suitable for CI research. The issues in CI use and adoption are complex, and require a qualitative approach to understand in depth. On the other hand, measuring the effects in practice allows for comparisons, which is valuable for verifying hypotheses, studying improvement actions and measuring costs. CI as a technical system provides a fruitful central point of measurement for a software project. This study contributes another example of mixed methods CI research. It is unique in utilizing an observational method as its main qualitative data collection method.

This study utilizes an expert analysis method of categorizing build failures within the CI system, through the use of a Jenkins plug-in. This allows the researcher to deepen the analysis of different kinds of failures by easily combining the categorizations with other CI data. Categorization within the CI system allows the developers themselves to record their analysis of failed builds while investigating their fixes. In their work, software developers rely on information from the CI system, mainly build logs, to analyze build failures. Therefore, the CI system is a convenient place also for recording the analysis of those failures.

The managerial contributions of this study are as follows. Adopting a new CI process into a large, existing project is difficult. Immediate improvements when adopting a new CI system should not be expected. Existing project practices, such as regarding software architecture or testing may need revising. How the technical environments are set up and maintained seems to be crucial. As a final observation, when people are faced with problems that are not caused by their own work or that they have little control over, frustration can cause them to circumvent or ignore these problems, rather than trying to fix them.

7.3 Future Research

This study raises some pointers for future research. The utility value of build failure categorization is an aspect that deserves further studies. Categorization of failures within the CI system allowed for quantifying some qualitative findings about experienced problems. As such the results gathered based on the categorization were a great benefit for us researchers. They were valuable for demonstrating where major problems in CI use lie. Unfortunately in this case, the developers categorized only a relatively small number of builds. For stronger results, the developers need a better intrinsic motivation to perform categorization.

Could manual build failure categorization be developed into a more useful tool for developers by providing more immediate information as feedback? In this study, results were calculated by hand, and were not constantly available to the developers in their freshest form. Because the categorizing was done within the CI system, it would be possible to calculate results similar to those of this study and present them to the developers interactively.

Comparing the effectiveness and accuracy of automated categorization against a lightweight expert analysis method is an interesting topic for future research. Automated analysis methods require no effort from the developers and can be performed on a large set of data. Because all builds can be easily analyzed with automated methods, they do not suffer of selection bias in choosing which builds to categorize. Also, the determinism of an automated approach guarantees a single interpretation viewpoint, whereas expert analyses by different developers may be slightly different. Automated methods may also provide more useful information during development: the same method that is used to detect a type of failure from a log file can be used to point out the problem to developers, aiding them in their troubleshooting. However, there may be limitations to the descriptive power of automated methods. They can only find problems that are clearly evident from the build log output. A manual approach allows the developer to investigate deeper issues, perhaps providing a more accurate root cause analysis regarding build failure.

And finally, an automated method to detect similarities between manually categorized failures of the same kind could be valuable. This could aid in generating rules for automated categorization, bridging the benefits of both kinds of categorization methods. Developers could be better motivated to manually categorize failures if those categorizations lead to later automatic detection of similar failures.

Bibliography

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000. ISBN 0201616416.

A. Begel and N. Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 255–264, September 2007. doi: 10.1109/ESEM.2007.12.

S. Bellomo, N. Ernst, R. Nord, and R. Kazman. Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 702–707, June 2014. doi: 10.1109/DSN.2014.104.

Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi: 10.1191/1478088706qp063oa.

Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57(0):21 – 31, 2015. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2014.07.009.

William G. Cochran. The $\chi 2$ Test of Goodness of Fit. *The Annals of Mathematical Statistics*, 23(3):pp. 315–345, 1952. ISSN 00034851. URL `http://www.jstor.org/stable/2236678`.

Duncan Coutts. The dreaded diamond dependency problem, April 2008. URL `http://www.well-typed.com/blog/9/`.

D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Global Software Engineering,*

2007. ICGSE 2007. Second IEEE International Conference on, pages 81–90, August 2007. doi: 10.1109/ICGSE.2007.13.

Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. Challenges when adopting continuous integration: A case study. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, volume 8892 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2014. ISBN 978-3-319-13834-3.

J. Downs, B. Plimmer, and J.G. Hosking. Ambient awareness of build status in collocated software teams. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 507–517, June 2012. doi: 10.1109/ICSE.2012.6227165.

John Downs, John Hosking, and Beryl Plimmer. Status communication in agile software teams: A case study. In *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, ICSEA '10, pages 82–87, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4144-0. doi: 10.1109/ICSEA.2010.20.

Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50 (9–10):833 – 859, 2008. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2008.01.006.

Christof Ebert, Casimiro Hernandez Parro, Roland Suttels, and Harald Kolarczyk. Improving Validation Activities in a Global Software Development. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 545–554, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. 2014.

Dale H Emery. Writing maintainable automated acceptance tests. In *Agile Testing Workshop, Agile Development Practices, Orlando, Florida*, 2009.

Martin Fowler. Continuous integration, May 2006. URL `http://martinfowler.com/articles/continuousIntegration.html`.

Martin Fowler. Eradicating non-determinism in tests, April 2011. URL `http://martinfowler.com/articles/nonDeterminism.html`.

D. Goodman and M. Elbaz. "it's not the pants, it's the people in the pants" learnings from the gap agile transformation what worked, how we did it, and what still puzzles us. In *Agile, 2008. AGILE '08. Conference*, pages 112–115, August 2008. doi: 10.1109/Agile.2008.87.

Timothy J Halloran and William L Scherlis. High Quality and Open Source Software Practices. In *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, 2002.

James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 85–95, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302455.

J.D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 29(6):481–494, June 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1205177.

Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, August 2010. ISBN 9780321601919.

Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. 2014.

Narendra Kurapati, Venkata Sarath Chandra Manyam, and Kai Petersen. Agile software development practice adoption survey. In Claes Wohlin, editor, *Agile Processes in Software Engineering and Extreme Programming*, volume 111 of *Lecture Notes in Business Information Processing*, pages 16–30. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30349-4.

Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The speedways and country roads to continuous deployment. 2015.

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635920.

A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130, April 2008. doi: 10.1109/ICST.2008.22.

A. Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE '08. Conference*, pages 289–293, August 2008. doi: 10.1109/Agile.2008.8.

Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 631–634, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2494580.

Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). pages 121–128. IEEE, August 2013. ISBN 978-0-7695-5076-3. doi: 10.1109/AGILE.2013.17.

H.H. Olsson, H. Alahyari, and J. Bosch. Climbing the "stairway to heaven"; – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 392–399, September 2012. doi: 10.1109/SEAA.2012.54.

Michael Q. Patton. *Qualitative Research & Evaluation Methods*. Sage Publications, 3rd edition, January 2002. ISBN 0761919716. Published: Hardcover.

Mary Poppendieck and Tom Poppendieck. *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

Mary Poppendieck and Tom Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Pearson Education, 2007.

Kaisu Rantanen. *Benefits of automated system-level regression testing; A case study*. M.sc. thesis, Aalto University, Espoo, 2010.

Pilar Rodríguez, Jouni Markkula, Markku Oivo, and Kimmo Turula. Survey on agile and lean usage in finnish software industry. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 139–148, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1056-7. doi: 10.1145/2372251.2372275.

Pilar Rodríguez, Kirsi Mikkonen, Pasi Kuvaja, Markku Oivo, and Juan Garbajosa. Building lean thinking in a telecom software development organization: Strengths and challenges. In *Proceedings of the 2013 International Conference on Software and System Process*, ICSSP 2013, pages 98–107. ACM, 2013. ISBN 978-1-4503-2062-7. doi: 10.1145/2486046.2486064.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. ISSN 1382-3256. doi: 10.1007/s10664-008-9102-8.

David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85, July 2004. ISSN 0163-5948. doi: 10.1145/1013886.1007523.

Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *IASTED Multiconferences-Proceedings of the IASTED International Conference on Software Engineering, SE 2013*, pages 736–743, 2013.

Daniel Ståhl and Jan Bosch. Automated software integration flows in industry: a multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 54–63. ACM, 2014a.

Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, January 2014b. ISSN 01641212. doi: 10.1016/j.jss.2013.08.032.

Dirk Stelzer and Werner Mellis. Success factors of organizational change in software process improvement. In *SOFTWARE PROCESS – IMPROVEMENT AND PRACTICE*, pages 227–250, 1999.

A. Strauss and J.M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 1998. ISBN 9780803959408.

Damian A. Tamburri, Patricia Lago, and Hans van Vliet. Organizational social structures for software engineering. *ACM Comput. Surv.*, 46(1): 3:1–3:35, July 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522971.

VersionOne. 8th annual state of agile survey, 2014. URL `http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf`.

Robert K Yin. *Case study research: Design and methods*, volume 5. Sage Publications, second edition edition, 1994.

Shin Yoo, Robert Nilsson, and Mark Harman. Faster fault finding at google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*, 2011.

Appendix A

Jenkins Plug-In for Categorizing Build Failures

A plug-in for the Jenkins CI system was developed during this study for manual categorization of build failure causes. Developers used the plug-in in the CI user interface to log failure causes and additional free text information. Figure A.1 shows the categorization view that the developers used.

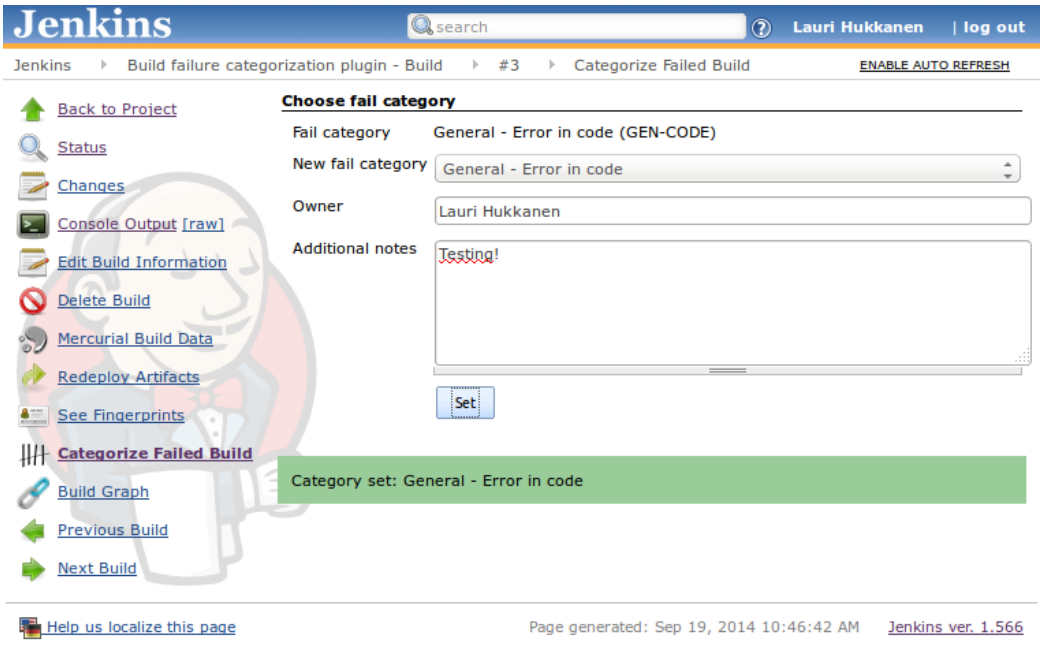


Figure A.1: Screenshot of the Jenkins plugin to categorize build failures, showing the input fields available.